DTIC FILE COPY

(2)

AD-A223 153

CECOM

CENTER FOR SOFTWARE ENGINEERING

ADVANCED SOFTWARE TECHNOLOGY

CLEARED
FOR OPEN PUBLICATION

DEC 20 1989 12

DIRECTORATE FOR FREEDOM OF INFORMATION
AND SECURITY REVIEW (OASD-PA)
DEPARTMENT OF DEFENSE

Subject: **Final Report - Real-Time Ada Problem Study**

DTIC
ELECTE
JUN 22 1990
S E D

CIN: **C02 092LA 0005 00**

24 MARCH 1989

895381

90 06 21 037

PREFACE

The real-time technology program at the Center for Software Engineering, CECOM, is based on recognized problems encountered in the development of embedded real-time Ada systems. The first step in the program was to define this set of root problems. The approach was to conduct interviews with both program managers and system developers working on Ada real-time applications for the Army and then to analyze, categorize, and enter into an database the resulting set of issues.

This document includes the two technical reports that resulted from the effort to define this set of problems. The authors were chosen because of their proven expertise in real-time development with Ada. They could enrich the results of the interview process with their own experience in this area.

The first report is entitled "Software Engineering Issues on Ada Technology Insertion for Real-Time Embedded Systems". LabTek Corporation, the author, had proven expertise in embedded system design utilizing Motorola MC680X0- based processors.

The second report is entitled "Software Engineering Problems Using Ada in Computers Integral to Weapons Systems." Its author, Sonicraft, had expertise in evaluating system requirements and in performing high-level system design utilizing the Intel 80X86 family of processors.

DTIC COPY INSPECTED 6

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

TABLE OF CONTENTS

SOFTWARE ENGINEERING ISSUES ON Ada TECHNOLOGY

INSERTION FOR REAL-TIME EMBEDDED SYSTEMS

FINAL REPORT

PREPARED FOR:

U.S. Army HQ CECOM
Center for Software Engineering
Advanced Software Technology
Fort Monmouth, NJ 07703-5000

PREPARED BY:

LabTek Corporation
853 Beechwood Road
Orange, CT 06525

DATE:

30 September 1987

Ada Technology Issues

EXECUTIVE SUMMARY

This report is the result of a study to identify and classify the problems associated with the current use of the Ada programming language, particularly in real-time embedded applications. The term "Ada technology" which appears in the report title refers to currently available Ada compilation systems, tools and associated methodologies. The conclusions of the study indicate that some of the prominent problems are due to the immaturity of the commercially available Ada compilation systems which includes the runtime environments. It is believed that these compiler problems will be nearly eliminated by the compiler vendors within five years, provided that the demand for solutions to these real-time problems is sustained. Other problems include the substantial resources needed to develop Ada software and the lack of Ada debugging tools. In addition, the inexperience with Ada imposes many management concerns. While trying to maintain cost and schedule objectives, management must train its personnel and deal with the unknowns of the chosen Ada compilation system.

Table of Contents

Table of Contents

Table of Contents

Ada Technology Issues

1. Overview

The programming language Ada, was developed to reduce the life cycle development cost of mission critical real-time embedded systems. Most of these systems are characterized by long development times and long lives with continual changes and expansions in their environment. The ability to evolve and grow is important. Development efforts are further complicated by the high reliability required for these systems. Building reliable software is a major issue when failures could result in loss of life and tremendous economic costs. Deterministic behavior is an essential criteria of an embedded real-time system because a random, non-repeatable failure may lead to a catastrophic life threatening situation. Here, the stability of the target system is of utmost importance.

Typically, embedded systems software is developed on large minicomputers with a variety of peripherals such as disk drives, printers, and a large number of interactive terminals. This system is often referred to as the "host" computer. The host computer used for software development is usually different from the embedded computer that will run the application programs. The computer hardware and operating system on which the application program executes is called the target system. The host's operating system, such as VMS[2] or UNIX[3] supports the tools needed to create the application program. The tools include editors for writing the programs, compilers and assemblers for translating the modules into executable code, and utilities for preparing the application for execution (among others).

There are a number of aspects of the Ada language that make it different from languages that have been used to implement real-time systems. Perhaps the most significant difference, from a real-time software point of view, is the interaction between the language features and the execution time support for those features. The most obvious feature that illustrates this point is the Ada *task*. Tasking has always been used in real-time systems, however it is normally provided by a separate "executive" that manages the tasks as separate cooperating programs. In Ada, tasks are part of the language, and therefore every Ada implementation must support tasking, and there is a high degree of integration between language features and the tasking model. This precludes using a typical independently developed executive to provide the tasking services. This tight coupling between the language and the routines that are necessary to support the runtime execution has a major impact on the flexibility of Ada applications. An Ada runtime (often called runtime support) is the set of procedures or functions required to support the code generated from a compilation. It is this runtime that is now provided by compiler vendors that was previously developed by application builders.

The use of Ada for real-time embedded systems is exposing a number of serious software engineering issues. It is apparent from interviewing program managers, software managers, software engineers, and application engineers that many are encountering similar types of problems when trying to apply the current commercially available Ada technology. The

---

[2]VMS is a trademark of Digital Equipment Corporation.

[3]UNIX is a trademark of AT&T Bell Laboratories, Inc.

purpose of this report is to identify the problems encountered by real-time embedded applications using Ada technology and to classify them.

2. Approach

The approach used to obtain the information in this report was to:

1. identify sources of information related to problems with the use of Ada,
2. contact those sources of input and obtain specific information on Ada issues,
3. verify the input obtained with an additional source, where necessary,
4. organize and categorize that information into common areas, and
5. analyze the resultant material for direction in the resolution of these issues.

Three primary sources of input were identified. The most important source of information was through interviews. Interviews were conducted with program managers, software managers and application engineers by LabTek personnel to discuss the problems that were encountered. The reported problems were analyzed by LabTek personnel and verified by an additional source (i.e. compiler vendors, Ada Joint Program Office (AJPO), etc.) where necessary. The Ada Runtime Environment Working Group (ARTEWG) and Special Interest Group on Ada (SIGAda) meetings served as a second source of information. These meetings and conferences were attended by LabTek personnel with special attention given to talks concerning Ada and real-time applications. Compiler vendor products, tools, and methodologies that support them were also reviewed at these meetings. The third source was the current Ada and software engineering related literature.

2.1 Interviews

LabTek Corp. has established a wide customer base and has been involved with providing consulting services for many customers on their Ada applications. In addition to conducting the following interviews, LabTek has included its own experiences in identifying the problems. Most of the projects interviewed below were still under development at the time of the interview.

2.1.1 Program Manager Interviews

- Advanced Field Artillery Tactical Data Systems (AFATDS), Fort Monmouth, NJ, CECOM
- Army Secure Operating System (ASOS), Fort Monmouth, NJ, CECOM
- Improved HELLFIRE DIGITAL AUTOPILOT (DAP), Redstone Arsenal, Alabama, MICOM
- Howitzer Improvement Program (HIP), Picatinny Arsenal, AMCOM
- Tank Improvements (M60A3), Picatinny Arsenal, NJ, AMCOM

* Nuclear, Biological, & Chemical Reconnaissance (NBCRS), Picatinny Arsenal, AMCOM

* Sense and Destroy Armor (SADARM), Picatinny Arsenal, NJ, AMCOM

* Single Channel Objective Tactical Terminal (SCOTT), Fort Monmouth, NJ, CECOM

2.1.2 Private Industry Interviews

* Advanced Field Artillery Tactical Data Systems (AFATDS), Magnavox Electronics Systems Co., Fort Wayne, Indiana

* Bradley Fighting Vehicle (BEDS), FMC/Digital Turret Distribution Box, San Jose, CA

* Bradley Fighting Vehicle (DTDB), FMC/Digital Turret Distribution Box, San Jose, CA

* Bradley Fighting Vehicle, GE/Digital Electronics Control Assembly, Pittsfield, MA

* F4-J Weapon System Trainer, Science Applications International Corp. (SAIC), Huntsville, Alabama

* Lightweight Helicopter (LHX), United Technologies Sikorsky Aircraft, Stratford, CT

* Tank Improvements (M60A3), Computing Devices Co., Ottowa, Canada

* Nuclear, Biological, & Chemical Reconnaissance (NBCRS), TRW, Redondo Beach, CA

2.2 ARTEWG and SIGAda Meetings

The ARTEWG is a group sponsored by SIGAda whose purpose is to address the problems encountered in runtime environments. Some of these problems are occurring because Ada compiler implementors are making executive function decisions that application developers previously made on their own when they built their own executives. The current implementations are not satisfying all embedded application needs. [4] It is ARTEWG's responsibility to establish conventions, criteria, and guidelines for Ada runtime environments. This will facilitate reusability and transportability of Ada program components, improve the performance of those components, and provide a framework which can be used to evaluate Ada runtime systems. Tom Griest of LabTek is an ARTEWG principal and the leader of subgroup one, the Implementation Dependencies Subgroup. As such, he participates in the ARTEWG meetings and provides input into its documents. These documents were examined for their relevance to this report.

The SIGAda meetings are held four times a year (three times in the U.S. and once in Europe) and LabTek personnel attend these conferences with special attention paid to talks by users concerning real-time embedded applications and methodologies. Vendor products are also examined.

2.3 Literature

The Ada issues database was obtained from the ARPANET. This database is an accumulation of problems and questions on Ada and coined "Ada issues". This database was scanned for the real-time relevant issues.

A literature search was also conducted. Sources included papers from the ARTEWG, ACM Ada Letters, ACM SIGPLAN Notices, ACM Software Engineering Notes, IEEE Software, IEEE Transactions on Software Engineering, Defense Science & Electronics and Computer Language magazine. All relevant articles were reviewed and evaluated.

2.4 Criteria for Issue Definition

To determine the validity of a reported issue two questions were asked: 1) Is the problem the result of the improper use of Ada?, and 2) Is the problem really a symptom of an underlying issue? If the answer to question one was affirmative the problem was rejected. If the answer to question two was affirmative the problem was restated to indicate the true problem. In addition, LabTek obtained multiple sources reporting the same problem. This can be seen in section 4, the Issue Vs. Source Matrix.

3. Ada Technology Issues In Real-Time Embedded Systems

Developing software for real-time embedded applications requires diverse skills and approaches for problem recognition and solution. The software implementation of a problem solution, however, can be approached by using a set of techniques that are application-independent. These techniques form the basis of a software engineering methodology. Software engineering is modeled on the time-proven techniques, methods, and controls associated with hardware development. Although fundamental differences do exist between hardware and software, the concepts associated with planning, development, review, and management control are similar for both system elements. The key objectives of software engineering are 1) a well-defined methodology that addresses a software life-cycle of planning, development, and maintenance, 2) an established set of software components that documents each step in the life-cycle and shows traceability from step to step, and 3) a set of predictable milestones that can be reviewed at regular intervals throughout the software life-cycle. [9]

Ada technology is used to produce the real-time software in an embedded system. Real-time software measures, analyzes and controls real world events as they occur. A real-time system must respond within strict time constraints. Note that this is different from "interactive" or "time-shared" where the response time can normally be exceeded without disastrous results. Ada technology consists of the compilation systems, tools, methodologies, principles and techniques that are employed to develop and maintain Ada software.

The problems encountered when using Ada technology for real-time embedded applications are enumerated here. The problems are divided into different classifications. In some cases, an issue may be applicable to more than one classification. When this has occurred the issue has been placed in the classification where it has the most significance. A tree of the different classifications can be seen in Figure 1. The tree provides the structure for section 3 of this report.

Ada Technology Issues

To provide easy extraction of problem definition from background and support, each problem is detailed with these four subheadings:

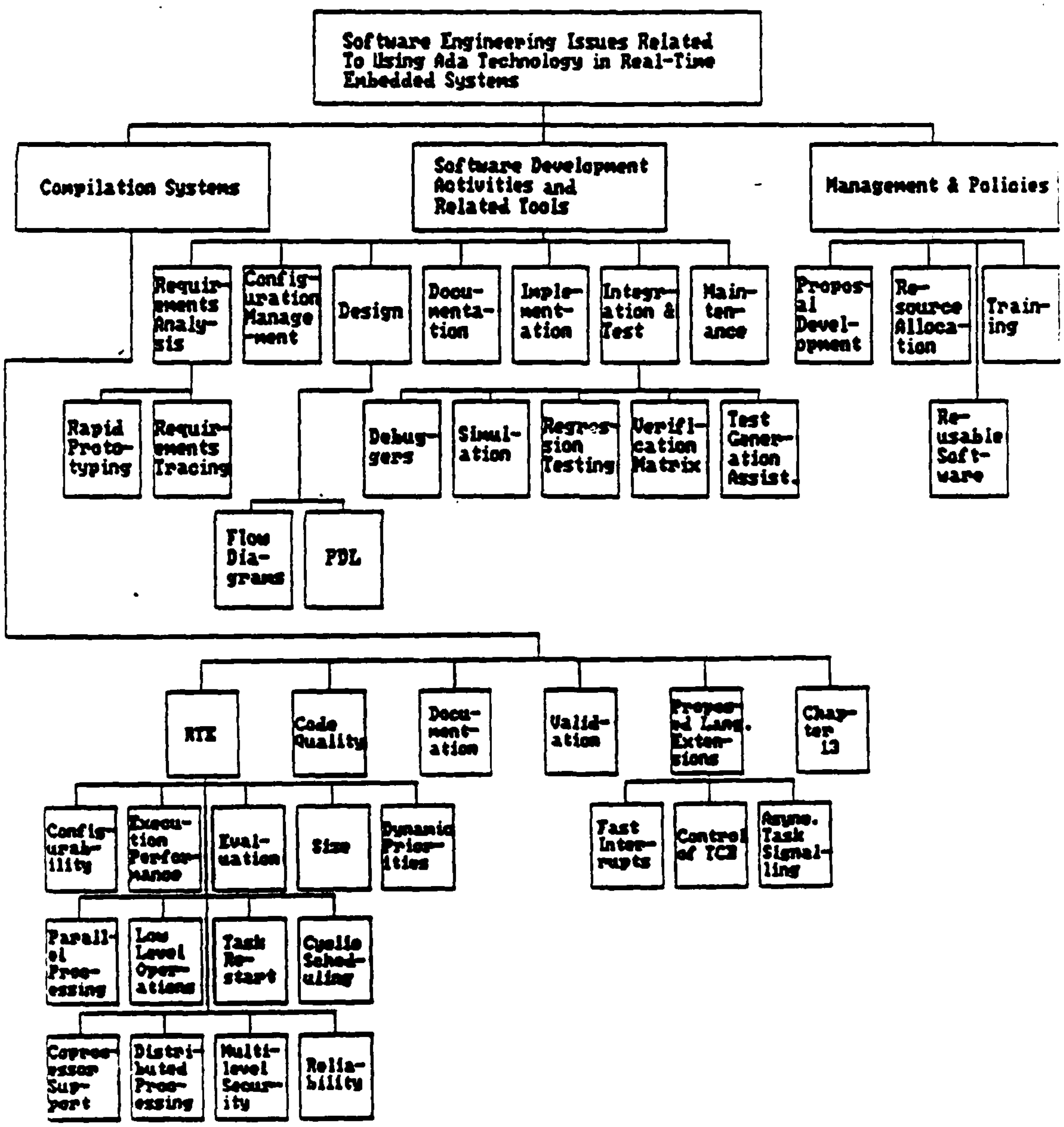
1.) Issue/Problem Definition - The "Issue/Problem Definition" subheading provides a conclusive problem definition (what is at issue). If the problem or issue manifests itself differently than the analyzed and defined (conclusive) problem, then a few sentences restating problem "symptoms" may be included. If there isn't any problem associated with a particular classification it will be so stated. In these cases, the classification was included for completeness to this report.

2.) Background - The "Background" subheading provides a framework for understanding both the general nature of the problem and the analysis that follows this part. Brief observations about the problem or issue categorization and a few sentence discussion of the source(s) of information about the problem will be included in this subheading, as well as any relevant definitions.

3.) Analysis & Support - The "Analysis & Support" subheading will contain detailed observations about the substance of the issue as well as any assumptions made by LabTek during the analysis. Also included in this subheading is any data or rationale that supports the findings and a summary of the conclusions about the problem. If available, any related or connected problems will be mentioned as well as any specific causes, conditions, or constraints under which the problem manifests itself. It there is other specific criteria which may be used by the reader to evaluate the validity of the conclusions, it will be provided.

4.) Problem Resolution - The "Problem Resolution" subheading may provide constructive and reasonable recommendations for problem resolution in the form of plans, approaches, methods, work-arounds, engineering techniques, etc.. It may detail associated conditions or constraints affecting a reasonable resolution. When a problem resolution is provided an associated time frame will be given. Within the context of this report, a "short-term" problem resolution is one that can be implemented within one to two years, and a "long-term" problem resolution is one that can be implemented in two to ten years.

Figure 1. Taxonomy of Softwaré Engineering Issues with Ada Technology



3.1 Compilation Systems

An Ada compilation system translates an Ada source program into its machine code equivalent. There are many aspects that make Ada different from other high-order languages (HOLs). The combination of features to support parallel execution, exception handling, information hiding, strong typing, etc., make it different. However, with respect to real-time embedded applications, the most significant difference is the inclusion of a tasking model and other features that were previously not part of other HOLs, but of a separate executive. Now, the Ada compilation system supplies the code that was previously provided by a separate executive. That is, Ada compilation systems provide an extensive "runtime" which other traditional compilers did not. The runtime (also called runtime support) is the set of procedures or functions required to support the code generated from a compilation (i.e. entry call in a task, exception raising, abort processing, string catenation, etc.).

The following sections discuss the problems with the currently available Ada compilation systems in detail.

3.1.1 Runtime Environments

Issue/Problem Definition: In other languages, application writers were accustomed to tailoring their executives so that they were extremely efficient for a particular application. Now, with the advent of Ada, the compiler implementors have control of the runtime environment, and that tailoring ability has been lost to a great extent in the current generation of Ada products.

Background: Applications are dependent on the runtime to provide an efficient implementation to support system services (i.e. tasking, storage management, exception handling, etc.). A runtime environment (RTE) includes all of the runtime support routines, the conventions between the runtime routines and the compiler, and the underlying virtual machine of the target computer. Virtual is used in the sense that is may be a machine with layered software (a host operating system). An RTE does not include the application itself, but includes everything the application can interact with. Each layer has a protocol between it and the layer underneath it for interfacing. In the event that there isn't any operating system layer, the runtime includes those low-level functions found in an operating system. See Figure 2.

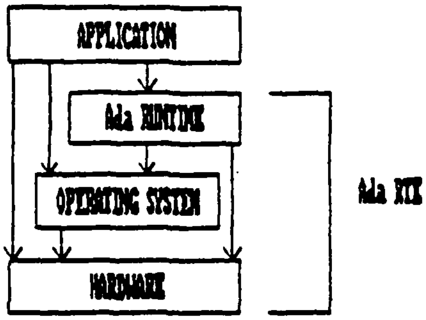


Figure 2. Ada Runtime Environment (RTE)

This information was supplied by the following sources: Project Interviews, ARTEWG & SIGAda Meetings, Current Software Engineering Literature, and LabTek Experience.

Analysis & Support: The complexity of the interface between the runtime and the generated code from the Ada compiler is substantial. The difficulty is in the lack of intimate knowledge of how the runtime links to the compiler generated code. Furthermore, the runtime must meet a rigorous set of tests to insure that it complies with the Ada standard. The runtime environment of the Ada compilation system must always comply with the rules of the Ada language as defined by the Reference Manual for the Ada Programming Language. These are major changes from the way other runtimes were developed in the past.

In summary, there is always a reluctance on the part of a programmer to give up control. The problem will become less severe as the compiler vendors provide greater flexibility in their runtimes and as software designers learn to take advantage of the features that are provided.

Problem Resolution: (Short Term) A list should be generated of what Ada runtime features are most needed by real-time embedded applications. This list could be provided to compiler implementors, and later used as a checklist by people evaluating compilers.

(Long Term) These features, if provided in a configurable runtime, could solve many of the real-time issues.

3.1.1.1 Configurability

Issue/Problem Definition: Ada runtime environments do not always provide sufficient configurability.

Background: Software is said to be "configurable" if the user can select various options when building the application software. In the case of the Ada runtime environment, a configurable component might be the type of scheduler used for the tasking algorithm.

This information was supplied by the following sources: Project Interviews, ARTEWG & SIGAda Meetings, Current Software Engineering Literature, and LabTek Experience.

Analysis & Support: There is a trend developing towards providing "intelligent" loading capabilities. An intelligent loader will only load those features of the runtime that are required by the application. That is, there is no point in loading the code to support tasking, if tasking is not used by the application. In a sense, this could be considered a configuration of the runtime. However, runtime configurability, in the context of this report, would allow an application engineer to specify such features as: the tasking algorithm used by the scheduler, or the memory management technique used, (among others). In conducting the various interviews, intelligent loading capabilities were appearing. However, when a special runtime configuration was needed (such as a particular type of scheduler), the contractor had to rely on the compiler vendor to perform the configuration.

In summary, this will be a more severe problem for some applications than others. A group of engineers trying to write an operating system in Ada will have extreme difficulty if not able to control the tasking aspects of the runtime. Compiler vendors are aware that users need configurable runtimes due to the requests they receive to tailor them.

Some people doubt whether complete "configurability" is even possible for real-time embedded systems and feel that some "hand tuning" will always be necessary. The general belief is that it is possible to achieve the necessary performance and adaptation requirements for most systems if sufficient configurability is provided.

Problem Resolution:

(Short Term)

1.) The ability for users to provide substitutes for any vendor supplied runtime routines is necessary. This is believed to be the only way that distributed systems can be developed, as well as many non-standard single-processor implementations.

2.) The vendors must supply sufficient interface information on their runtime routines to allow customers to make these substitutions.

3.) Schedules must include time for this customization process.

(Long Term)

4.) Vendors should be encouraged to provide configurability that has a high degree of automation. This configuration process must be able to be performed by the application engineers, since relying on the compiler vendor often results in an

unacceptable delay. An .bility to selectively choose from among several versions of the runtime routines to build a runtime environment that is customized for a particular application is what is needed. In this way, the optimal tasking algorithm (dynamic priorities, run-till-blocked, round-robin, time slicing, etc.) can be chosen. Menu oriented systems that verify compatibility are probably best. Provisions must be made with the configuration management mechanism to allow determining the components of a generated runtime.

5.) Support tools to make the configuration process semi-automatic and reliable.

Issue/Problem Definition: A configurable math library is required by many projects, and is not provided. Along with the library, a set of tests should be provided to verify the accuracy of the library after changes have been made.

Background: Accuracy and timing vary greatly from application to application. For example, some projects use a look-up table for speed in computing transcendental functions.

This information was supplied by the following sources: Project Interviews, ARTEWG & SIGAda Meetings, Current Software Engineering Literature, and LabTek Experience.

Analysis & Support: A mechanism is needed which allows users to modify the standard math library provided by vendors. (Note: the Ada language does not include facilities for transcendental math functions.) In talking with the various engineers it was pointed out that many projects have their own math library.

In summary, many applications custom code their own math library. This practice should be changed.

Problem Resolution: (Short Term) Ada math libraries are being developed and placed into the public domain. Provisions must be made for making application builders aware of this resource.

3.1.1.2 Execution Performance

Issue/Problem Definition: The quality of the compiler generated code is often inadequate for real-time embedded systems.

Background: Efficiency is used in this context to mean the combined performance of the compiler generated code and the runtime. It is a relative term, comparing the processing rate of the high level code to the hypothetical optimal assembly language.

Timing is a critical constraint in a real-time application. The sampling of data, a calculation providing input to a feedback loop, the driving of servos, the handling of external interrupts are some examples of functions that must be performed in a precise time interval. If they are not performed in the specified time interval the system will not function properly. In a flight system this problem could be manifested in a pilot receiving the wrong positional information at a critical time. Although the runtime can create short term processing delays, the average throughput of the computer is generally most impacted by the optimization of the generated code.

This information was supplied by the following sources: Project Interviews, ARTEWG & SIGAda Meetings, Current Software Engineering Literature, and LabTek Experience.

Analysis & Support: Which Ada features are most costly to use is a function of each implementation. Examples of critical areas where the runtime code has to be efficient are:

- The Ada rendezvous (task signalling and communication). In general, the task management function tends to have the greatest amount of overhead.

- Dynamic storage management.

- Exception handling. Often the problem is avoiding overhead in the absence of exceptions. Since every subprogram invocation involves a new exception scope, care must be taken to reduce the overhead associated with maintaining the current appropriate exception handler. Tradeoffs can be made between exception handling that propagate exceptions quickly, or those that do not impose delays during normal (non-exception) processing. Most exceptions occur only when severe problems (hardware or design) exist, provided the exception mechanism isn't misused. Therefore the penalty for slow exception handling is usually acceptable if non-exception processing can execute without additional overhead.

- Constraint checking is another candidate, depending on the implementation. When constraint checking becomes too severe, most implementations have a mechanism to disable the constraint checks. Ada provides a standard technique for this in the SUPPRESS pragma.

- Support for interrupts.

- Procedure call overhead.

- Stack overflow check.

- Array referencing.

In summary, eighty one percent of the projects interviewed reported that the quality of the generated code was a major problem for their real-time embedded application.

Problem Resolution: (Long Term) Vendors should be encouraged to provide a high degree of optimization. This encouragement can take the form of a policy to interpret the language standard so as to permit optimizations where they are currently limited by language rules. Compilers must also be improved to take advantage of the interaction of the suppress pragma and optimization. That is, the rules requiring the proper point of raising exceptions restrict some optimizations. If the checks are suppressed by a pragma, these additional optimizations can take place.

Another solution might be to subset use of language features and optimize for that subset.

3.1.1.3 Evaluation

Issue/Problem Definition: It is difficult to evaluate Ada runtime environments for suitability in an application.

Background: Evaluation is the ability to determine whether or not the runtime environment under consideration will be adequate for a particular application.

This information was supplied by the following sources: Project Interviews, ARTEWG & SIGAda Meetings.

Analysis & Support: Often the compiler user's manual is the only material available to lend insight into the intricacies of the compiler. Frequently it is inadequate for detailed evaluation.

An area where this is especially true is in measuring the dynamic memory usage. One of the requirements the US Army places on its contractors is that 50% of the total memory be reserved for expansion. Since Ada allocates memory dynamically, it is often difficult to insure that the worst case will not use some of the reserved 50% of memory. It should be possible to compute the memory usage, but the documentation for many Ada implementations do not provide sufficient detail to allow this. The use of dynamic memory allocation cannot be avoided simply by eliminating the use of Ada "allocators" in the source code. Ada compilation systems use dynamic memory allocation/deallocation for many different operations, from allocating space on the stack for subprogram parameters to using heap storage for manipulation of unconstrained arrays. Some tasking implementations require dynamic storage allocation to provide a separate task stack and task control block that is allocated during task elaboration. Therefore the user may not know what features use dynamic storage allocation, and it may change in future releases of the compiler system.

In summary, often the only means of evaluating a compiler before purchase is through the documentation and talking with the compiler vendor. The details of how the compiler implements the semantics of the Ada code must be made known to the application developers in order for them to assess the impacts and tradeoffs of using various Ada language constructs. In general, very little performance data is currently available for Ada compilation systems. The best method for determining the critical timing parameters and storage usage is to analyze the code of the runtime routines. Ideally the vendor should perform this analysis and provide the information to perspective customers. In practice, partially due to the frequent changes made to the runtimes, the vendors do not have this information available. The users are often forced to purchase the compiler and source code of the runtime in order to complete their evaluation.

Problem Resolution: (Short Term) Provide users with an accurate estimate of the overhead an capabilities associated with the various Ada implementations. This would require that a study on RTEs be performed and benchmarks run for evaluation. The options available in terms of configurability should be included, with sizes listed and limitations imposed on the application when selecting the different configurations. For example, if restricted tasking service is selected, the application may be limited to using only simple rendezvous capability with no parameters supported. In particular, the functionality and size of a minimal RTE should be specified.

(Long Term) Benchmarks must be developed that indicate critical timing issues of Ada implementations. These are extremely difficult and time consuming to generate. For this reason, it is important that an effort be supported to develop these tests and to make them available to the Ada community. Ideally, each compiler that is suitable for embedded processor use could be evaluated each year after it validates, and its evaluation placed in the public domain. The activities of the Ada Compiler Evaluation Capability (ACEC)

effort and the Performance Issues Working Group (PIWG) of SIGAda should be monitored, and if appropriate, supported.

3.1.1.4 **Size**

Issue/Problem Definition: Runtime environment (RTE) sizes are often too large.

Background: Size, in this context, refers to the memory capacity required to accommodate the runtime.

This information was supplied by the following sources: Project Interviews, and LabTek Experience.

Analysis & Support: Some applications may have a total memory capacity of only 8K bytes. With the US Army requirement that 50% of total memory be reserved for future expansion, this leaves 4K bytes available for the RTE and the application code. It is currently not technically possible to put the RTE and the application code in 4K bytes. The smallest RTE encountered was 6K bytes, but smaller RTEs are gradually appearing.

A question that many people ask is "Why is the RTE so big?". The pressure on the vendors has been to validate early, rather than optimize the runtimes. Also, the effort required to produce small runtimes is significant. The trend now is to address these issues and to reduce the runtime size. The competition in the Ada compiler market is forcing changes to occur.

Another concern relating to configurable runtimes is "When does one know how big the total runtime is going to be?". The exact answer will not be determined until the implementation of the application is completed. However, a fairly accurate estimate is needed before development begins in order to insure that the system has sufficient memory capacity. It should be possible to make a more accurate estimate when the vendors provide documentation about their configurable runtimes. Still, a significant amount of the application will need to be known (e.g. what Ada features are used) prior to developing an accurate estimate.

In summary, sixty-nine percent of the projects interviewed reported that the runtime required a substantial percentage of the total system memory thus severely limiting the memory available for the application code.

Problem Resolution: (Short Term) A partial solution is to use a customized RTE at the expense of not being able to use full Ada (i.e. text_io, generics, tasking and dynamic allocation). Restricting the use of Ada features may also limit the ability to reuse previously developed software.

It has been suggested that for applications with very stringent memory requirements, a stripped down version of Ada be used so that the runtime is minimal. This combined with a heavy reliance on assembler code would allow very small applications to be written partially in Ada. The benefit of this approach is questionable. The application would probably be better written entirely in assembler rather than a mixture of Ada and assembly language when the amount of Ada code is small, the runtime has been virtually eliminated, and extensive use of assembler is made.

Issue/Problem Definition: Memory is a critical constraint in many real-time embedded applications.

Background: Although the current trend is to think that memory is inexpensive and can be added if additional memory is needed, this is not the case for some embedded applications.

This information was supplied by the following sources: Project Interviews, and LabTek Experience.

Analysis & Support: Following are reasons why memory is a constraint in some systems:

- Use of Single Board Computers (SBC). A SBC consists of a processor, memory, and I/O ports on the same board. The advantage to having on-board memory is that signals do not have to be sent over the backplane, thus reducing the time to read and write the memory. This can significantly increase performance.

- Packaging. If additional memory is required, it is not always feasible to add an additional memory board. In many cases there is no room to add another board, or there are weight and power considerations. Replacing the existing memory chips with denser memory chips does not always solve the problem either. Special requirements are placed on systems when memory must be radiation hardened. Density and cost of this type of memory, which is necessary in many military applications, do not lend themselves to "just adding more".

- Single Chip Computers. Use of Large Scale Integration (LSI) single chip computers often restricts memory to less than 4KB of ROM and 256 bytes of RAM. These processors increase reliability and reduce costs in areas where they can support the processing required. Other benefits include lower power consumption and reduced weight and size.

In summary, fifty-six percent of the projects interviewed reported that memory was a constraint in their system. Although hardware engineers can provide numerous reasons why additional memory cannot be added (e.g. increased weight, increased power consumption, board layout modifications, cost, retrofits), these reasons can no longer be accepted if Ada is to be used in all weapon system software.

Problem Resolution: (Short Term) Hardware designers should be made aware of the directive to use Ada in weapon systems, and include sufficient memory in their designs to accommodate Ada.

(Short Term) An investigation into the expansion of memory for embedded computers is needed. Information about optimal memory technologies has to be provided to the hardware designers.

Another solution might be to subset use of language features and optimize for that subset.

3.1.1.5 **Dynamic Priorities**

Issue/Problem Definition: There isn't any provision for dynamic priorities of tasks in the Ada language.

Background: A task is said to have a "dynamic priority" if its priority level is changeable at execution time. Some applications have the need to dynamically alter the priority of a currently running task.

This information was supplied by the following sources: ARTEWG & SIGAda Meetings, and LabTek Experience.

Analysis & Support: Ada does not support a capability for dynamically altering the priority of a currently running task. The value for the PRAGMA PRIORITY is static and therefore cannot be changed at runtime. Implementations may support an alternate set of priorities that control tasking in the case where the Ada PRIORITY is identical or undefined. That is, if two tasks have the same priority, the implementation is free to schedule them in any order. This allows an implementation defined subpriority, which may be dynamic, to control the scheduling. This capability is not supported by many implementations, and a standard does not exist to help provide commonality.

In summary, dynamic priorities are required by some real-time applications, and they are not provided for by the Ada language. The user can implement a limited form of dynamic priorities with considerable effort.

Problem Resolution: (Long Term) The areas of the language that do not directly support application requirements should be evaluated to determine if a consistent approach to support these requirements is possible. If these services can be provided in an acceptable manner without a language change, then this is the desired approach. However, in some cases it may be desirable to make small, compatible changes in the language in order to accommodate these requirements in a more efficient and consistent fashion.

(Long Term) Changes to the standard for the Ada Language (ANSI/MIL-STD-1815A) that would alleviate problems encountered by real-time embedded systems should be considered. Although making specific changes is beyond the scope of this study, a possible candidate for evaluation is "Dynamic Priorities".

3.1.1.6 **Parallel Processing**

Issue/Problem Definition: Current RTE's do not support parallel processing.

Background: Most signal processing functions require parallel data-path computers.

This information was supplied by the following sources: ARTEWG & SIGAda Meetings, and LabTek Experience.

Analysis & Support: The Reference Manual for the Ada Programming Language, section 4.5(5), states that operands of an expression "are evaluated in some order that is not defined by the language". This appears to exclude parallel execution, as would be the case in a MIMD (multiple instruction multiple data) machine. Confusion is provided by the note in the Reference Manual, section 9(5), that talks about the flexibility to use multiple processors when the same effect is guaranteed. However, this is just a note and not technically part of the standard.

Further clarification is provided in the "Rationale for the Design of the Ada Programming Language." In section 3.8, "Assignment Statements - The Ada Model of Time", on page 29

paragraph 7, it states: "Note finally that whenever order is not defined, the reference manual uses the phrase 'in some order that is not defined', rather than the phrase 'in any order'. The intent of the chosen wording is to leave the order undefined but nevertheless require that it be done in some order, and thus EXCLUDE PARALLEL EVALUATION". [emphasis added].

The Rationale continues with the reason for this restriction. This issue and others similar to it, notably in the constraint checking/handling areas, seem to be overly restrictive with respect to parallel architectures. Specifically, single instruction multiple data (SIMD), multiple instruction multiple data (MIMD), and massively parallel processors (MPP), will be extremely limited if relaxation of this interpretation is not forthcoming.

Problem Resolution: (Long Term) The areas of the language that do not directly support application requirements should be evaluated to determine if a consistent approach to support these requirements is possible. If these services can be provided in an acceptable manner without a language change, then this is the desired approach. However, in some cases it may be desirable to make small, compatible changes in the language in order to accommodate these requirements in a more efficient and consistent fashion.

(Long Term) Changes to the standard for the Ada Language (ANSI/MIL-STD-1815A) that would alleviate problems encountered by real-time embedded systems should be considered. Although making specific changes is beyond the scope of this study, a possible candidate for evaluation is "Parallel Processing".

3.1.1.7 **Support of Low Level Operations**

Issue/Problem Definition: Ada does not provide a mechanism to control the processor state (including interrupt masks required for critical sections).

Background: Although Ada provides a mechanism to directly manipulate memory mapped hardware, no capability exists within the language to access internal processor registers. Such a mechanism would be difficult to standardize across many architectures.

This information was supplied by the following sources: Project Interviews, and LabTek Experience.

Analysis & Support: Provision should be made in implementations so that users can control the processor state and not interfere with the runtime system. For example, some implementations require that the application run in a non-privileged state (USER MODE). This is not always acceptable, and simply executing an assembly language routine to change to SYSTEM MODE will not solve the problem. Changing the processor state from USER to SYSTEM, needs to be done in conjunction with the runtime. Since stacks used for different states are often separate, simply changing state will result in an error condition. Also, subsequent calls to the runtime (possibly due to exceptions) are likely to cause unpredictable results.

In summary, in real-time programming there is frequently a need to enable and disable interrupts which is performed by setting or clearing interrupt masks. It is easy for a programmer to write an assembly language routine to manipulate an interrupt mask and call this routine from an Ada program. The problem occurs because the assembly language routine is not working in conjunction with the runtime environment provided.

Unpredictable results could occur. The way for the user to manipulate interrupt masks (and processor state) without interfering with the runtime is to have the compiler vendors supply a routine to do this that is compatible with their runtime environment.

Problem Resolution: (Long Term) Provision should be made in implementations so that users can control the processor state, and not interfere with the runtime system.

(Long Term) The areas of the language that do not directly support application requirements should be evaluated to determine if a consistent approach to support these requirements is possible. If these services can be provided in an acceptable manner without a language change, then this is the desired approach. However, in some cases it may be desirable to make small, compatible changes in the language in order to accommodate these requirements in a more efficient and consistent fashion.

(Long Term) Changes to the standard for the Ada Language (ANSI/MIL-STD-1815A) that would alleviate problems encountered by real-time embedded systems should be considered. Although making specific changes is beyond the scope of this study, a possible candidate for evaluation is "Support of Low Level Operations".

3.1.1.8 **Task Restart**

Issue/Problem Definition: Applications which require that a separate thread of control (task) be restarted at the beginning after being interrupted part way through have difficulty mapping this requirement to Ada.

Background: An example where a task restart would be needed is in the following situation: Suppose that an aircraft was on a mission to deliver a weapon to a specified target. The pilot flies within the vicinity of the target. The flight control software activates an Ada task which calculates the current aircraft position as well as the target position. Before this task can complete, a task of higher priority becomes active, a task to defend the aircraft against incoming threats. The pilot performs the necessary defensive maneuvers and once the threat is no longer a threat, the pilot resumes with the mission to deliver the weapon to the target. However, the aircraft is no longer at the same position that the previously suspended Ada task recorded. If the suspended Ada task became active now, the positional information would be wrong and would lead to disastrous results. What is needed here is the ability to restart this task.

This information was supplied by the following sources: ARTEWG & SIGAda Meetings.

Analysis & Support: The obvious question is: is this really a requirement, or simply the implementation of a requirement? After careful scrutiny, it appears that certain applications do have a need to be able to have multiple tasks, where one task might be preempted by a higher priority task, and the result of the preemption is to make the continuation of the preempted task meaningless. The standard Ada solution to this problem is to ABORT the preempted task, and then re-activate a new task. This creates a few undesirable side effects, not the least of which is likely to be unacceptable performance degradation.

In summary, there are real-time applications which have a need for a task restart capability.

Note: Preliminary Ada (May 1979) provided for an exception "FAILURE" to be raised within one task from another task. This would support the desired effect of "Task Restart", however the feature was removed from the language prior to the 1980 version.

Problem Resolution: (Long Term) The areas of the language that do not directly support application requirements should be evaluated to determine if a consistent approach to support these requirements is possible. If these services can be provided in an acceptable manner without a language change, then this is the desired approach. However, in some cases it may be desirable to make small, compatible changes in the language in order to accommodate these requirements in a more efficient and consistent fashion.

(Long Term) Changes to the standard for the Ada Language (ANSI/MIL-STD-1815A) that would alleviate problems encountered by real-time embedded systems should be considered. Although making specific changes is beyond the scope of this study, a possible candidate for evaluation is "Task Restart".

3.1.1.9 **Cyclic Scheduling**

Issue/Problem Definition: There is no explicit provision for cyclic executives in the Ada language.

Background: A cyclic executive runs on a scheduled time basis and is either difficult or impossible to achieve with accuracy on many implementations.

The

```
        loop
          -
          -
        end loop;
```

structure is not sufficient since it is not periodic. If different control paths are taken within the loop, or if other tasks (possibly interrupt level) preempt execution, then this loop will not execute in constant time.

This information was supplied by the following sources: ARTEWG & SIGAda Meetings, and LabTek Experience.

Analysis & Support: The Ada language can support some degree of periodic processing by using the DELAY statement. Although some implementations provide a reasonable mechanism for this, the DELAY statement is not always adequate for this application. (Note: many people are concerned with the wording in the Reference Manual for the Ada Programming Language regarding the accuracy of the delay in a DELAY statement: "...for at least the duration specified by the resulting value". This is not the problem, although it can contribute to it.) Assuming that an implementation has infinite accuracy for type DURATION and DELAYs exactly that amount, there is still a problem specifying the duration for a cyclic task. The problem is that the duration value is a delay from the current time, not a fixed interval. Therefore, the clock must be read and the cycle computed in the simple_expression allowed for the delay statement. However, there is no way to insure that an interrupt (and possibly a higher priority task) is not executed between the time the clock is read in the simple_expression and when the delay duration is actually interpreted by the

runtime. This allows the execution of the cycle to begin with possibly twice as much jitter as what would normally be expected due to higher priority processing.

In summary, cyclic executives are an important part of real-time embedded software. How they are implemented in Ada is something that requires careful consideration.

Non-Solutions: An implementation could provide a pragma that would insure that the expression in the following DELAY statement is implemented as non-interruptible, thus resolving this issue, but at the price of being rather messy (not recommended).

Another non-solution is the use of interrupt tasks, and using a hardware interval timer to provide the periodic scheduling. This will work in some applications, but for those applications that have several different periods, this quickly becomes unmanageable.

Problem Resolution: (Long Term) The areas of the language that do not directly support application requirements should be evaluated to determine if a consistent approach to support these requirements is possible. If these services can be provided in an acceptable manner without a language change, then this is the desired approach. However, in some cases it may be desirable to make small, compatible changes in the language in order to accommodate these requirements in a more efficient and consistent fashion.

(Long Term) Changes to the standard for the Ada Language (ANSI/MIL-STD-1815A) that would alleviate problems encountered by real-time embedded systems should be considered. Although making specific changes is beyond the scope of this study, a possible candidate for evaluation is "Cyclic Scheduling".

3.1.1.10 Floating Point Coprocessor Support

Issue/Problem Definition: There is a lack of a standard for floating point coprocessor support. Some compilers require a floating point chip to perform floating point calculations, other compilers can't utilize the chip if present.

Background: A floating point coprocessor is a high-performance numerics processing element that extends the main processor architecture by adding significant numeric capabilities and direct support for floating-point, extended integer, and BCD data types.

This information was supplied by the following sources: Project Interviews, and LabTek Experience.

Analysis & Support: Applications should have the option to run with or without a coprocessor. All compilers should provide floating point capabilities independent of whether the floating point processor is configured into the system or not.

The presence of a floating point chip would increase performance in a real-time embedded application that required floating point operations to be performed. On the other hand, an application may not be able to repackage to include a floating point chip on the board and this application could utilize a software floating point emulation.

The use of tasking introduces additional overhead when the floating point chip is present. The floating point coprocessors contain internal registers which must be saved and restored during task switches. Ideally, intelligent context switch software (or hardware) would only

save and restore these registers when a context switch occurs to a task that uses floating point, instead of when any context switch occurs.

Also, the support for the intrinsic functions such as sine and cosine are important. These functions are not part of the Ada language and need not be supplied by an Ada compiler vendor to validate. A numerics working group exists within SIGAda to attempt to standardize the interface to common numeric functions. For the greatest amount of efficiency, it is usually best that the vendor supply these functions as a special package. This allows the implementation to take advantage of the hardware that is available. When a math coprocessor is supported, it should be used to implement these functions, since their calculation is often greater than fifty times faster using the hardware coprocessor. With proper compiler optimization and use of the INLINE pragma, it can be possible to achieve nearly optimal use of floating point accelerators even with user written packages.

In summary, the decision to use the floating point chip should be up to the application programmer and the Ada compiler should perform properly with or without the presence of the floating point chip.

Problem Resolution: (Short Term) The evaluation of compilers should include a section regarding the support level for coprocessors.

3.1.1.11 **Distributed Processing**

Issue/Problem Definition: There are no commercially available Ada implementations whose runtime environments support distributed computer configurations that operate over communication links.

Background: A distributed system is a configuration of processors, memories, and links in which each processor has a designated local memory that it can access in significantly less time than it can access either shared memory or the local memory of other processors. Typically, such systems have a high bandwidth communication link between processors but little or no shared memory. The key to efficient use of these systems is to structure the system and distribute the work load so as to limit the interprocessor communication. [14]

This information was supplied by the following sources: Project Interviews, ARTEWG & SIGAda Meetings, Current Software Engineering Literature, and LabTek Experience.

Analysis & Support: Progress has been made in the area of shared-memory multiprocessor implementations of Ada, however the technology is closely coupled to the hardware configuration and cannot be easily migrated to other targets. Currently, distributed processor software is most often being developed as individual Ada programs with custom communication services rather than as monolithic Ada programs.

In summary, there is no common approach to developing runtime environments, methodologies, and tools that support the distribution of Ada program entities across multiprocessor configurations or distributed networks. [4]

Problem Resolution: (Long Term) Distributed processing, in the sense that Ada tasks from a single program are distributed across several processors connected by a communication link, is an area that still requires research and development. Solving the problems of communication errors, possible failure of a processor, and implementing efficient

"SELECT" statements for the Ada rendezvous will require more time and effort before the results are available to the general public.

3.1.1.12 Multi-level Security Support

Issue/Problem Definition: Ada runtime environments do not currently support multi-level security.

Background: Security is the protection of computer hardware and software from accidental or malicious access, use, modification, destruction, denial of service, or disclosure. [11]

This information was supplied by the following sources: Project Interviews, ARTEWG & SIGAda Meetings, and Current Software Engineering Literature.

Analysis & Support: The vast amount of software being developed is for large distributed battlefield computer systems. These systems tend to involve many different aspects of battle management ranging from tracking and launching to maintaining personnel records. Likewise, the operators tend to have different security requirements. The need for the software to be built on top of a secure operating system stems from these conditions. An officer who needs only to assess the current ammunition reserves should not be able to launch weapons.

In summary, without providing a secure base upon which the Ada application runs, security in an integrated battlefield system cannot be insured. It is possible to develop an Ada application layered on top of an existing secure operating system, but available secure operating systems are currently not very suitable for most real-time embedded applications.

However there are many small projects that are unlikely to require multi-level security since they are stand-alone processors with a limited number of functions, all at the same security level.

Problem Resolution: (Long Term) Like Distributed Processing above, the area of security will require time and effort before small, efficient, secure runtimes are available for use on real-time embedded systems. If this area is a priority for the military, then it will probably require special funding. Since the perception of a weak market for secure embedded systems exists, commercial compiler vendors will need encouragement to devote resources to this effort.

3.1.1.13 Reliability

Issue/Problem Definition: With respect to software reliability, Ada systems are too new to provide any significant data as to whether programs written in Ada are more reliable than those written in other languages. Reliability in Ada systems is not seen as a problem at this time, and this section was only included for completeness.

Background: Reliability is the ability of an item to perform a required function under stated conditions for a stated period of time. Software reliability is the probability that software will not cause the failure of a system for a specified time under specified conditions. It's the ability of a program to perform a required function under stated conditions for a stated period of time. [11]

This information was supplied by the following sources: Project Interviews, ARTEWG & SIGAda Meetings, and LabTek Experience.

Analysis & Support: As experience with Ada technology increases, more information will become available as to the reliability of the systems developed in Ada. It is felt that the strong typing mechanism of Ada is extremely valuable with respect to reliability. This mechanism insures that all variables are identified and typed and assignments are checked to be valid at execution time.

Due to the immarturity and frequent updates for Ada compilers, a short-term reliability problem may be found with some implementations because of errors with the compilation and/or runtime systems.

Problem Resolution: Since reliability does not present a problem, there is no problem resolution required at present.

Issue/Problem Definition: With respect to supporting hardware reliability, Ada does not address the reliability issue with any specific language constructs.

Background: CPU fault tolerance is the built-in capability of a system to provide continued correct execution in the presence of a limited number of hardware or software faults. [11]

This information was supplied by the following sources: Project Interviews, ARTEWG & SIGAda Meetings, and LabTek Experience.

Analysis & Support: Ultra high reliable systems require that the software continue to operate in the presence of CPU faults. Although this may seem impossible, careful analysis indicates that many faults (for example nuclear EMP) are momentary and do not result in permanent interruption in processing capability. However, it is essential that the program be able to recover from such faults and continue execution from the last check point. In ballistic missile systems check points may be as close as a few instructions in order to minimize the disruption. The program must constantly be storing multiple copies of check point addresses and data used in computations. Ada does not directly support the ability to recover from such CPU faults.

In summary, Ada does not support hardware reliability with specific language constructs. Exception handlers are provided to detect software faults, but in rare cases may also be used to detect and recover from hardware faults.

Problem Resolution: (Long Term) Reliability in this context is specific to recovery from CPU faults. Although it is possible for an Ada program to checkpoint its data, due to the complexity of program elaboration, it would be difficult with an off-the-shelf runtime to roll-back and recover from a CPU reset. Unlike assembly language routines where data is largely statically allocated, Ada's dynamic nature of data makes reconstruction much more difficult. Research and development needs to be done to resolve the impact of using Ada in the most critical applications areas where extended interruption of service is unacceptable.

3.1.2 Code Quality

Issue/Problem Definition: The quality of the generated code, in terms of being optimal, was a major problem for many real-time embedded systems.

Background: The quality of the generated code has a tremendous impact on the ability to use a compiler for real-time software production. Ideally, the code generated by the compiler would be nearly as efficient as what an experienced assembly language programmer could produce. This has always been an elusive goal, however it is not unheard of to have some programs developed in a high level language perform better than their assembly language counterparts. This is generally due to an improved algorithm that was practical to implement because of the high level language.

This information was supplied by the following sources: Project Interviews, ARTEWG & SIGAda Meetings, Current Software Engineering Literature, and LabTek Experience.

Analysis & Support: In Ada, optimization is very important in order to eliminate unnecessary runtime checking. Although the runtime checks can be suppressed, it is often desirable to leave them on to detect latent software faults, or even some hardware faults.

Subprogram invocation is another very important area for optimization. Due to the strict rules of passing parameters, elaborating and initializing local objects, and entering new exception scopes for each subprogram call, it is essential that this mechanism be kept as efficient as possible.

There is no substitution for having optimized code. The difference between optimization and no optimization can be as much as 15 to 1 or greater in performance, although typical numbers are much smaller. The excuse that computer hardware is much faster than it used to be, and therefore code efficiency is no longer as important, is only partially true. The efficiency that can be sacrificed so that a high order language can be used varies tremendously from application to application. In general, knowing the code quality of the proposed compilation system should be a major concern, so that the appropriate speed processor can be selected for the application.

In summary, eighty-one percent of the projects interviewed reported that the quality of the generated code was a major problem for real-time embedded applications.

Non-Solution: When confronted with a large overhead in subprogram invocation the alternative is to force programmers away from the modularity of subprograms and back into straight-line coding practices. In some cases the pragma INLINE can be used to generate the subprogram statements in line at the point of call, and to avoid much of the overhead associated with the calling sequence. However, this can become impractical when calls are made to the subprogram from many points in the program due to the large expansion in code size.

Non-Solution: The measure of last resort is to fall back to assembly language in the most time critical aspects of the program. This works very well for applications that are large, but have a small real-time component. All too often however, programs tend to have time critical portions spread throughout. In these cases, the high level code portion looks like initialization code that calls the main assembly language program (not recommended).

Problem Resolution: (Long Term) The area of code quality can be resolved by applying commercial incentives on vendors improving their products. This can be accomplished by implementation of the recommendation specified under section 3.1.1.3.

3.1.3 Documentation

Issue/Problem Definition: Areas lacking in compiler vendor documentation are details concerning: 1) critical timing parameters, and 2) the runtime environment.

Background: In this section, "documentation" refers to the compiler documentation.

This information was supplied by the following sources: Project Interviews, and LabTek Experience.

Analysis & Support: Sixty-three percent of the projects interviewed reported that inadequacies were encountered with the compiler documentation. This was particularly evident in the area of critical timing parameters (i.e., task context switch time, synchronous rendezvous time, complex rendezvous time, and interrupt latency time). Values for times, such as interrupt latency due to disabled interrupts within the runtime are usually not provided in the standard documentation (and sometimes impossible to obtain). These values are needed in order to compute what variance can be expected with the DELAY statement and other timing issues. It is extremely difficult to use benchmarks to measure this type of timing. Benchmarks tend to show typical or average execution times. What is often more important is the worst case time.

This issue should not to be confused with the information supplied in Appendix F of the compiler documentation entitled "Implementation-Dependent Characteristics", as specified by ANSI-MIL-STD-1815A-1983. Appendix F specifies: 1) The form, allowed places, and effect of every implementation dependent pragma. 2) The name and the type of every implementation-dependent attribute. 3) The specification of the package SYSTEM. 4) The list of all restrictions on representation clauses. 5) The conventions used for any implementation-generated name denoting implementation-dependent components. 6) The interpretation of expressions that appear in address clauses, including those for interrupts. 7) Any restriction on unchecked conversions. 8) Any implementation-dependent characteristics of the input-output packages. [2]

The best method for determining the critical timing parameters is to analyze the code within the runtime. The disadvantage is that often this information is needed in helping to select the compiler, and often it is not provided until the selection has been made.

An example of an inadequacy in the runtime environment section is that it is difficult to measure dynamic memory usage on some implementations. This is usually because the documentation does not provide adequate details on how the runtime uses dynamic storage.

In summary, the current compiler documentation is lacking in the areas of critical timing parameters and runtime environment algorithms. This information is necessary to be able to evaluate the compiler before selection and should be provided.

Problem Resolution: (Short Term) By including the quality and completeness of the compiler documentation in the evaluation of compilers, this will allow users to select the implementations that have the level of documentation that is desired.

3.1.4 Validation

Issue/Problem Definition: In general, the compiler implementor's thrust has been towards validation of their compilers at the expense of poor optimization of its generated code, configurability of its runtime environment, or implementation of machine dependent features.

Background: Validation is the process of checking the conformity of an Ada compiler to the Ada programming language and of issuing certificates indicating compliance of those compilers that have been successfully tested. [1] It should be emphasized that the intent is only to measure conformance with the standard. Any validated compiler may still have bugs and poor performance, since performance is not being measured by the validation tests. [6]

To obtain a validation certificate a compiler implementor must exercise an Ada Compiler Validation Capability (ACVC) test suite. The current level is Version 1.9 and it contains a series of over 2500 tests designed to check a compiler's conformance to the DoD's Ada language standard, ANSI/MIL-STD-1815A-1983. To date, compiler implementors have been very concerned with obtaining the status of "validated" for their compilers. Having a validated Ada compiler is no longer just a marketing ploy, it is required by the DoD that contractors developing Ada software must use a validated Ada compiler (DoD Directive 3405.2).

This information was supplied by the following sources: Project Interviews, and Current Software Engineering Literature.

Analysis & Support: With the initial validation phase completed for most compilers, the compiler implementors are shifting their emphasis to concentrate on improving the efficiency of the generated code (code optimization) and providing more user configurability of the runtime environment.

Validation issues have confused many application builders. Questions have arisen as to whether or not similar, but not identical, hardware that was used for validation can be considered validated. Other issues have appeared involving what constitutes "maintenance" of a compiler, and how much of it can undergo change and still retain validation status. Also, since Ada validation status is only retained for one year after validation, concerns have been expressed for programs that do not want to change the version of their compiler after they begin testing. New policies have been developed to support baselining a compiler with respect to a project, and deriving validation status for similarly configured machines. Although there are still unresolved issues, the associated problems are minor and are unlikely to adversely impact any development programs.

In summary, the real-time issues of performance, support for low-level operations and decreasing runtime environment sizes have taken a back seat to obtaining a compiler validation certificate.

Problem Resolution: (Short Term) The ACVC must be expanded to include all of the Chapter 13 features. Information on the new validation policy should be provided to all PMs and government contractors using Ada.

3.1.5 Proposed Ada Language Extensions

In interviewing various engineers the following language features were desired but not present in the current Reference Manual for the Ada Programming Language: 1.) Support for fast interrupts, 2.) Greater control of the Task Control Block (TCB), and 3.) Asynchronous Task Communications. These features are explained further below.

3.1.5.1 Fast Interrupts

Issue/Problem Definition: Fast interrupts are not explicitly supported by Ada.

Background: Support for fast interrupts would allow the application engineer to specify that an interrupt task will not require the runtime to perform a full context switch on interrupt entry (i.e. the interrupt task executes in the context of the currently active task). This is necessary in order to provide immediate response to hardware events.

This information was supplied by the following sources: Project Interviews, ARTEWG & SIGAda Meetings, and LabTek Experience.

Analysis & Support: Some compiler implementations provide this capability with an implementation dependent PRAGMA.

Problem Resolution: (Short Term) The suggestion is to make this a standard PRAGMA.

(Long Term) Changes to the standard for the Ada Language (ANSI/MIL-STD-1815A) that would alleviate problems encountered by real-time embedded systems should be considered. Although making specific changes is beyond the scope of this study, a possible candidate for evaluation is "Fast Interrupts".

3.1.5.2 Greater Control of the Task Control Block (TCB)

Issue/Problem Definition: Application engineers are not able to manipulate the TCB in most implementations.

Background: The task control block contains such information as: task priority, amount of time allocated to a time slice, status of the task, etc..

This information was supplied by the following sources: Project Interviews, and LabTek Experience.

Analysis & Support: The application engineer would benefit if able to manipulate the task control block. This is not a severe problem for most applications. The applications which require this capability are the ones that are trying to write operating systems in Ada. An operating system makes extensive use of tasking and the ability to specify the time-slice period, among other task characteristics, is desirable.

Problem Resolution: (Long Term) Changes to the standard for the Ada Language (ANSI/MIL-STD-1815A) that would alleviate problems encountered by real-time embedded systems should be considered. Although making specific changes is beyond the scope of this study, a possible candidate for evaluation is "Control of Task Control Block".

3.1.5.3 Asynchronous Task Communications

Issue/Problem Definition: The Ada rendezvous model uses a synchronous mechanism to communicate between tasks. Many applications require that a signalling task not be delayed until the signalled task is ready to accept the signal.

Background: The mechanism used to communicate between tasks in the Ada rendezvous model is that both tasks must be synchronized together before any data or control information can be transferred. This is *dissimilar* to the conventional mailbox techniques utilizing P and V semaphores to provide inter-task communications.

This information was supplied by the following sources: Project Interviews, ARTEWG & SIGAda Meetings, and LabTek Experience.

Analysis & Support: The argument for the Ada model is that it provides a unified approach for both data communication and generating events (signalling). However, many applications require that a signalling task not be delayed by waiting while the signalled task is not ready to rendezvous.

Problem Resolution: (Work-Around) The Ada solution to this issue is to place an intermediate task between the signalling task and the waiting task. This intermediate task would always be ready for a rendezvous and would effectively buffer the transaction to effectively provide asynchronous communications. The impact is to create an additional (logical) context switch. In the absence of an optimization which would eliminate the need of this additional context switch, the approach is not suitable for time critical applications. The combination of supporting fast interrupt tasks and asynchronous signals makes this optimization difficult for general use.

(Long Term) Changes to the standard for the Ada Language (ANSI/MIL-STD-1815A) that would alleviate problems encountered by real-time embedded systems should be considered. Although making specific changes is beyond the scope of this study, a possible candidate for evaluation is "Asynchronous Task Signalling".

3.1.6 Chapter 13

Issue/Problem Definition: Many of the features in Chapter 13 are not implemented in current commercially available compilers today.

Background: Chapter 13 of the Reference Manual for the Ada Programming Language is titled, "Representation Clauses and Implementation-Dependent Features". The Ada Compiler Validation Capability (ACVC) test suite does not thoroughly test features contained in this chapter. These features are optional and therefore a compiler can have the status of "validated" without any of these features implemented. However, many people feel that Chapter 13 is required for real-time embedded applications.

This information was supplied by the following sources: Project Interviews, ARTEWG & SIGAda Meetings, Current Software Engineering Literature, and LabTek Experience.

Analysis & Support: There is an inherent dilemma in the design of a high-order language with a systems programming capability. On the one hand the language designers are trying to achieve reliability by raising the level of the language. For example, they provide data

types and encourage the taking of an abstract view of objects, in which they are known only by the set of operations applicable to them: controlling the applicable operations enables incorrect usage to be detected. On the other hand, systems applications require the ability to stay rather close to the machine, and not only for reasons of efficiency. For example, defining a hardware descriptor must be done in terms of the physical properties such as the bit positions. A mapping different from that prescribed by the hardware would not merely be inefficient - it would be incorrect and would not work. To produce a correct program in such cases the application engineers are forced to abandon the abstract view and to work in terms of the physical representation. This contradiction cannot be avoided. The language must deal with objects at two different levels, the logical and the representation level [3]. That is the purpose of Chapter 13 of the Reference Manual.

In summary, fifty percent of the projects interviewed reported that a lack of Chapter 13 features implemented in their version of the compiler posed a serious problem.

Problem Resolution: (Short Term) As mentioned above under Validation, Chapter 13 features must be tested by the ACVC. Consideration should be given to making the Chapter 13 features mandatory for compilers used for embedded systems development.

3.2 Software Development Activities and Related Tools

Software development of a real-time embedded application is essentially comprised of three major phases - a concept definition phase, a development phase, and a deployment and operational phase. In the concept definition phase (the initial or early life-cycle phase), requirements are identified, an intended performance envelope is stated and statements of needs are written. The development phase primarily consists of the design, code and test activities concerned with the system implementation. The deployment and operational phase consists of the important maintenance and support activities required to complete the final or last phase of the life-cycle. [10]

It is important to point out that although the life-cycle phases imply that when one phase ends the next phase begins, in practice, there is substantial iteration between the phases. For example, if in the implementation phase a discrepancy is apparent, then it becomes necessary to go back to the design phase and make any modifications that are needed.

Following is a general problem resolution which covers the area of software development and related tools. It can be applied to most of the subsections under section 3.2.

Problem Resolution: (Long Term) The proper selection and use of tools is the greatest hope for improving the quality and lowering the cost of developing software. Although "software engineering methodologies" are conceptually the answer to many development problems, experience has shown that it is the tools that drive how engineers develop software. If a methodology is embodied within a tool that is perceived as being useful, then the methodology will be followed. In the absence of such tools, to get large groups of engineers to all use the same methods is nearly impossible. For this reason, the development of public domain software tools should be supported to the maximum degree possible. Already the WIS (WWMCCS Information System) program has supported the development of a set of tools through the Naval Ocean Systems Center. These tools have been placed in the Ada repository on the MIL-NET at SIMTEL20 and are available to US corporations. Although these tools are not yet production quality, there are efforts underway to fund their upgrading and maintenance. This activity should receive the full support of the services. It

may be possible to set up a program whereby tools that are taken from the repository and improved by users can be repurchased by the government to place the improved version back into the repository. This would provide the best of both "commercial off the shelf" and "government ownership" approaches to providing software for government projects.

The following sections apply to the general problem resolution of tool development:

reference section 3.2.1, "Requirements Analysis"
reference section 3.2.1.1, "Rapid Prototyping"
reference section 3.2.1.2, "Requirements Tracing"
reference section 3.2.2, "Configuration Management"
reference section 3.2.3, "Design"
reference section 3.2.3.1, "Flow Diagrams"
reference section 3.2.3.2, "Program Design Language (PDL)"
reference section 3.2.4, "Documentation"
reference section 3.2.5, "Implementation"
reference section 3.2.6, "Integration"
reference section 3.2.6.1, "Debuggers"
reference section 3.2.6.2, "Simulation"
reference section 3.2.6.3, "Automatic Regression Testing"
reference section 3.2.6.4, "Test Verification Matrix"
reference section 3.2.6.5, "Test Generation Assistance"
reference section 3.2.7, "Maintenance"

3.2.1 Requirements Analysis

Issue/Problem Definition: Several tools exist to help identify the requirements of a software system. However, these tools do not always integrate well with the other support tools for requirements tracing and configuration management.

Background: Requirements are precise statements of need intended to convey understanding about a desired result. They describe the external characteristics, or visible behavior of the result, as well as constraints such as performance, reliability, safety and cost. Analysis is the systematic process of reasoning about a problem and its constituent parts to understand what is needed or what must be done. [8]

The requirements analysis process defines the system's functional requirements (functions, inputs, outputs), external interface requirements (e.g. user interface), and performance, and other requirements such as security. [13]

This information was supplied by the following sources: ARTEWG & SIGAda Meetings, Current Software Engineering Literature, and LabTek Experience.

Analysis & Support: Some of these tools require specialized hardware to support graphics interfaces which are used to help visualize the structure of the software. Often the tools are cumbersome to use, and because no standard exists, training is a major problem with using the tools and the documentation that they generate. These problems are largely language independent. Therefore the only impact related to Ada is the trend towards standardization.

In summary, the capturing of requirements is language independent. The problems arise because there are tools for each aspect of software development and these tools are not integrated to work well together.

Problem Resolution: (Long Term) If tools can be developed in Ada for use in many development environments, then the availability of common requirements definition tools is a likely by-product.

Also see the general problem resolution under "Software Development Activities and Related Tools" above.

3.2.1.1 **Rapid Prototyping**

Issue/Problem Definition: The Ada language does not directly support rapid prototyping.

Background: Software prototyping is a process (the act, study, or skill) of modelling user requirements in one or more levels of detail, including working models. Project resources are allocated to produce scaled down versions of the software described by requirements. The prototype version makes the software visible for review by users, designers and management. This process continues as desired, with running versions ready for release after several iterations. [15]

This information was supplied by the following sources: ARTEWG & SIGAda Meetings, Current Software Engineering Literature, and LabTek Experience.

Analysis & Support: Ada is designed to support large applications where the interfaces between the modules must be rigidly defined in advance of their use. Ada forces users to "think before they code", which is beneficial for application code development, but not always expedient for rapid prototyping. When Ada interfaces change, the impact ripples back to all units using those interfaces. This makes down-stream changes to the code less than "rapid". Since the nature of rapid prototyping is to flush out different design ideas, the design and code tends to iterate on a week by week basis.

In summary, the rapid prototyping that is discussed here is the "throw-away" type. It is generally used to obtain information on system performance,or to test certain design ideas, and is discarded after it has served its purpose. As pointed out above, this process may not be as rapid in Ada as some other languages.

Problem Resolution: (Work-Around) The overhead in modifying the Ada interfaces must be reduced in order to prevent this activity from consuming an inordinate amount of time. Clever reuse of packages, especially generics may help alleviate this problem to some degree. Fast, interpretative Ada environments can also make the design changes less painful. Since rapid prototypes tend to be small in comparison to the application, the difficulties in using Ada to do rapid prototyping can be managed.

Also see the general problem resolution under "Software Development Activities and Related Tools" above.

3.2.1.2 **Requirements Tracing**

Issue/Problem Definition: Requirements tracing is not new to Ada, and is only included here for completeness.

Background: Requirements tracing is the practice of providing a documented allocation of requirements from the highest level specification down to the code section that satisfies those requirements. This is done to provide a measure of assurance that all of the requirements have been met and tested. It is also useful when requirements change and it becomes necessary to re-evaluate portions of a program for impact. If requirements are eliminated, then those code sections may be able to be reduced or removed entirely. In cases where new requirements are added or "clarified", the related code sections can be found quickly (in theory) and be updated.

This information was supplied by the following sources: ARTEWG & SIGAda Meetings, Current Software Engineering Literature, and LabTek Experience.

Analysis & Support: The standardization of Ada as a PDL (program design language) and as an implementation language does make tool generation to support requirements tracing through these phases more practical. Standard requirements keywords within the PDL can serve multiple purposes. If the C5 level specifications are automatically generated from the PDL this provides an input into that documentation. Since the Ada code can be included within the same file as the PDL, the same keywords provide traceability down to the code sections.

In summary, requirements tracing can be facilitated by Ada. If Ada is used as the PDL and as the implementation language, the generation of this requirements tracing tool becomes easier.

Problem Resolution: (Long Term) Work needs to be done to help link the high level specifications to the PDL.

Also see the general problem resolution under "Software Development Activities and Related Tools" above.

3.2.2 Configuration Management

Issue/Problem Definition: Difficulties exist because of a lack of commonality of tools.

Background: A configuration item is a collection of hardware or software elements treated as a unit for the purpose of configuration management. Configuration Management (CM) is the process of identifying and defining the configuration items in a system, controlling the release and change of these items throughout the system life cycle, recording and reporting the status of configuration items and change requests, and verifying the completeness and correctness of configuration items. [11]

This information was supplied by the following sources: Current Software Engineering Literature, and LabTek Experience.

Analysis & Support: Tools that process Ada source, such as the Ada compiler or PDL processor frequently maintain their own library of compilation units. This library usually contains a variety of file formats, and problems may arise when attempting to place the library under configuration management. Since the library can theoretically be regenerated

by reprocessing the Ada source code, typically only the source code is placed under configuration management. This is generally satisfactory provided that a mechanism exists to verify that the same command line options are used to invoke the tools. Ideally the entire library should be placed under CM for each major release.

Some tools are only available on graphics-based workstations, while the compiler may only be available on a mainframe computer. Maintaining consistent versions of all development materials can be extremely tedious under these circumstances. Some tools allow conversion of internal file formats to less efficient printable ASCII formats. This facilitates placing all configuration items into a common, machine readable database.

In summary, tools for different phases of the software life-cycle are often made by different vendors, so they are not usually integrated with each other. This has always been a problem.

Problem Resolution: See the general problem resolution under "Software Development Activities and Related Tools" above.

3.2.3 Design

Issue/Problem Definition: Designing a system in Ada is not seen to be a problem. However, the time required to design a system in Ada is significant and must not be overlooked (especially if training of personnel is needed).

Background: Design is the process of applying various techniques and principles for the purpose of defining a device, a process, or a system in sufficient detail to permit its physical realization.

This information was supplied by the following sources: Project Interviews, and LabTek Experience.

Analysis & Support: Ada facilitates the design process. Once the design is captured in an Ada PDL (Program Design Language), Ada consistency checking helps to verify the meeting of design requirements.

Problem Resolution: (Short Term) The design phase can be improved by the following recommendation: Managers should modify their schedules to allow more time in the beginning of the development phase for software definition. Ada designs generally develop slowly but the coding and debugging phases tend to proceed more smoothly (provided the tools are adequate). Also, there is a need to provide better guidelines for use of an Ada PDL.

(Long Term) There is still a need for design methodologies and tools that help to capture software requirements in a format suitable for an Ada implementation and to propagate these requirements in a form suitable for input into the PDL processor. One of the most difficult areas in systems design of DoD projects is the constant changing of requirements. Tools are necessary to help lessen the impact of this problem in Ada software systems.

Also see the general problem resolution under "Software Development Activities and Related Tools" above.

3.2.3.1 **Flow Diagrams**

Issue/Problem Definition: Problems associated with generating and interpreting flow diagrams are not unique to either Ada or real-time programs, and are only included here for completeness.

Background: Flow diagrams are used to graphically depict the data flow and control flow of programs. These graphic representations help convey the operation of programs so that high level interpretations can be made in a relatively short time.

This information was supplied by the following sources: ARTEWG & SIGAda Meetings, Current Software Engineering Literature, and LabTek Experience.

Analysis & Support: Flow diagrams are mentioned here for completeness, and to suggest that the transportability of Ada based tools make the development of a tool that could generate flow diagrams more cost effective. Although flow diagrams are most useful during the concept definition phase of software development, they are often helpful during subsequent phases for documentation purposes. Unfortunately, due to the difficulty in generated these diagrams, they are all too frequently not kept up to date with the design and code.

In summary, Ada can facilitate the generation of flow diagram tools, due to its transportability.

Problem Resolution: (Long Term) The standardization of Ada may make it practical to build a tool that would automatically generate various levels of flow diagrams from the Ada source code. This would provide the most accurate source of graphic documentation and would benefit those maintaining the programs.

Also see the general problem resolution under "Software Development Activities and Related Tools" above.

3.2.3.2 **Program Design Language (PDL)**

Issue/Problem Definition: PDL issues are not unique to real-time applications, or Ada, except for the possible concern about restricting some Ada features in the PDL because of execution time performance penalties.

Background: Program design languages are used to provide a high level representation of a program in a more rigorous fashion than either natural language or flow diagrams. With the adoption of Ada as the standard basis for PDLs, the additional features of being able to verify interfaces and showing dependencies, become available simply by processing the PDL with an Ada compiler.

This information was supplied by the following sources: Project Interviews, ARTEWG & SIGAda Meetings, Current Software Engineering Literature, and LabTek Experience.

Analysis & Support: The conflict arises when Ada is being used for both the PDL and the implementation language. There is a concern that if the designer stays at a high level (as they should) in the PDL, that the implementor will use an inappropriate construct in the code. This is further complicated if the PDL and code are never separated, but remain

within a single file. An example is that record types with default discriminants may be ideal for being able to change the representation of input buffers, however, due to the performance issues associated with this construct, it is probably not ideal for the implementation code.

Problem Resolution: See the general problem resolution under "Software Development Activities and Related Tools" above.

3.2.4 Documentation

Issue/Problem Definition: Ada tools such as PDL processors claim to support automatic generation of some specifications and software documentation. To a large degree, the output of these tools is somewhat less than satisfactory with respect to being complete.

Background: Software documentation is technical data or information, including computer listings and printouts, in human-readable form, that describe or specify the design or details, explain the capabilities, or provide operating instructions for using the software to obtain desired results from a software system. [11]

This information was supplied by the following sources: ARTEWG & SIGAda Meetings, Current Software Engineering Literature, and LabTek Experience.

Analysis & Support: The output of these tools does have the benefit of accurately tracking the PDL, and thus the implementation. This is important since when changes occur in the implementation, they are much more likely to be reflected in the C5 level specifications. The quality of the output of these tools is highly dependent upon the correct wording of PDL. Training is necessary to assist PDL developers so that the output is complete and consistent.

Problem Resolution: (Long Term) As always, the ability to generate quality graphics and text is important to the comprehension of design information. Many tools can support large documents, or graphics, but not both. Even when they support both, they may not work well with the configuration management tools. Since Ada is being recommended as the standard PDL, this helps to lend support to tool development to process PDL for documentation support. Work is needed to provide integration among interacting tools.

Also see the general problem resolution under "Software Development Activities and Related Tools" above.

3.2.5 Implementation

Issue/Problem Definition: The improper use of Ada features can create performance problems in an implementation.

Background: The implementation phase is the period of time in the software life cycle during which a software product is created from a design document and debugged. [11] Although the diagram of a software life cycle commonly shows phases following sequentially from each other, in actuality it is usually necessary and desirable to have some iteration between them. In addition, there is often significant overlap between phases. [13] A good background in software engineering practices is probably necessary to use the full

capabilities of the language - simply teaching professional programmers Ada is not enough. [12]

This information was supplied by the following sources: Project Interviews, and LabTek Experience.

Analysis & Support: Although support tools are needed in all phases of software development they are discussed briefly under "Implementation" because historically this is the phase that has been most influenced by tools. Support tools include the compilation systems, editors, assemblers, linkers, locators, loaders, librarians, pretty-printers, etc., that allow an engineer to input source code and generate executable machine code. Software support tools are also needed for debugging, configuration management, documentation generation, PDL processing, rapid prototyping, simulation and test support. The acquiring of a good toolset is important for the success of a project, and can reduce software development time significantly. Editors, assemblers, linkers, locators, loaders and librarians have been available for sometime and do not present substantial problems. The problems lie in the immaturity of Ada compilation systems, and the lack of debuggers and other support tools for them.

In summary, when implementing an Ada design for a real-time embedded system, care should be taken to insure that the appropriate Ada constructs are used, and that the code generated by the implementation does not require excessive time to execute.

Problem Resolution: (Short Term) It is hoped that a software engineer will study the problem and language constructs sufficiently before implementation to decide what features of the language should be used to best handle the problem at hand.

Also see the general problem resolution under "Software Development Activities and Related Tools" above.

3.2.6 Integration

Issue/Problem Definition: No problems specific to Ada have been identified in this area and this section has been included only for completeness.

Background: Integration is the process of combining software elements, hardware elements, or both into an overall system. [11]

This information was supplied by the following sources: Project Interviews, Current Software Engineering Literature, and LabTek Experience. Reference section 4 for the Issue Vs. Source Matrix.

Analysis & Support: In Ada, integration generally proceeds more rapidly partially due to its strong typing mechanism. Many errors that typically would not be caught until execution time in other languages are often caught at compilation time in Ada. [7]

Problem Resolution: See the general problem resolution under "Software Development Activities and Related Tools" above.

3.2.6.1 Debuggers

Issue/Problem Definition: Current in-circuit emulators operate (at best) as symbolic debuggers, and have no provisions to support real-time monitoring of dynamically allocated variables.

Background: Debugging refers to the process of removing errors from computer programs. Although debugging can and should be an orderly process, it is still very much practiced as an art. A software engineer, evaluating the results of a test, is often confronted with a symptomatic indication of a software problem. That is, the external manifestation of the error and the internal cause of the error may have no obvious relationship to one another. The poorly understood mental process that connects a symptom to a cause of a software problem is debugging. [9]

A debugging approach can be supplemented with debugging tools (often called debuggers). Debuggers are available in a wide variety, the most useful being dynamic debugging aids and automatic test case generators. Memory dumps and cross reference maps also aid the debugging process.

This information was supplied by the following sources: Project Interviews, and LabTek Experience.

Analysis & Support: An in-circuit emulator (ICE) allows code to be traced as it is executed without interfering with the real-time nature of it. That is, an ICE is a hardware substitution of the processor which supports real-time monitoring of the processor's activities. Emulators generally have capabilities to halt execution on user-programmable events, such as a write to a specified memory address. When the event occurs, a halt is generated, and the user has the ability to examine the previous instructions that lead up to the event (trace-back). This type of feature makes it possible to detect design errors that cause data to be corrupted by incorrect memory writes.

One of the side benefits of emulators is their ability to emulate the memory of the embedded system, and thus relieve the necessity to continually program new PROMs (Programmable Read-Only Memories). The emulator uses its read/write memory to substitute for the PROM, thus making the process of loading a program for execution much faster. Emulators also provide capabilities to profile program execution, which aids in determining which code sections should be optimized.

Current emulators operate (at best) as symbolic debuggers. This implies that the user has the ability to access certain "symbols" such as variable names, subprogram names, etc.. The symbol tables are generated by the compiler or assembler and provided to the emulator, usually along with the binary code through a down loading process over a data link. What is missing is the ability to operate on a source code level with emulators. Ideally, the user could specify a source statement line number at which the program should halt, and when the program does halt, have the same naming conventions and scope (possibly extended) of the program at that point in its execution. Single stepping of the program at the Ada source level is an extremely useful feature when trying to isolate difficult design errors. Often the hardware is new, and it operates either incorrectly, or in a way that is not reflected in the documentation. By stopping program execution at the points at which it interacts with the hardware, the user can quickly locate causes of failure. By not supporting source level operation, the Ada programmer is often forced to study the generated assembly language

code in order to resolve where to set break points, or to examine memory. This can be extremely tedious in situations where extensive optimization has been done on the generated code. Register assignments are not always obvious and reordering of the code can be very confusing.

Software debuggers are tools that support debugging without the aid of external hardware. Software debuggers on the other hand, do insert code into the executable module to produce a debugging version. Thus the debugging version and the real-time version are not the same executable modules. The overhead of a software debugger is usually not incurred with an ICE. Several manufacturers now provide powerful software debuggers that provide source level debugging. The application code is loaded into the target along with a debug support module. This module provides a communications interface to the host computer and the capability to set break points and examine memory. The user communicates directly to the host computer via a standard video terminal. The host computer is in turn connected to the target system. All communications between the user and the target are routed through the host computer debug support software. This allows the user interface to be very complex, including source level single stepping, yet the software loaded into the target is very simple, keeping it small. The main drawback is the loss of a trace-back and setting break points on certain types of events, such as memory writes to specific locations, for real-time execution.

In summary, what is desired for debugging is an in-circuit emulator with the features of advanced software debuggers, such as source level operations.

Problem Resolution: (Long Term) An emulator that supports a source level debugger would provide the ability to trace a particular variable, procedure, etc., in real-time. Tools to support in-circuit emulation are provided by third party vendors (i.e. not the compiler vendor), and the changes made in compilers are not supported immediately by the third party vendors. Hardware changes in the emulators may be necessary to support the dynamic nature of objects in Ada. For example, setting break points normally requires specifying a particular address (or range of addresses) with the emulator. However, since storage for data within procedures is allocated and deallocated dynamically at execution time of the procedure, it is impossible to know the exact memory address of the data prior to invoking the procedure. What may be required is the ability to establish an "arm" capability which detects entering a subprogram, and can calculate a data frame offset address at that time and set a break point based on that address. When the subprogram is exited (for any reason) the break point must be disarmed, since the memory address will be reused by the next subprogram.

The next generation of microprocessors appear to be helping to resolve this problem. They expect to be operating above the 30MHz mark where providing emulators is unlikely to be practical. The industry speculation is that new processors will contain on-chip logic to provide some of the features currently provided by emulators. In these environments, the debuggers are likely to be software driven.

Also see the general problem resolution under "Software Development Activities and Related Tools" above.

3.2.6.2 Simulation

Issue/Problem Definition: Simulation poses no new additional problems with respect to the Ada language. It is included here only for completeness.

Background: In software test environments, simulation is used to substitute for the real-life environment in which the application will operate. Often, various elements of the hardware are not available at the beginning of software/hardware integration, and it is necessary to simulate not only the external events, but pieces of the system as well. One of the advantages in using simulators is that usually they provide additional control over the environmental events that can not be achieved with the real system. For example, it may be impossible to test a helicopter at Mach 4, however simulators may allow the generation of data and events to test the software under the conditions that would be present if the helicopter could achieve Mach 4 flight. Simulators also assist in support of automatic testing. Programmable simulators can be set up to provide a scenario, and test support equipment can collect the responses of the software being tested. In this way many tests can be run in a much shorter time than what would be required to run real operational tests.

This information was supplied by the following sources: Project Interviews, and LabTek Experience.

Analysis & Support: The obvious drawback of simulators is the accuracy to which they can simulate the real world events. Many simulators have the capability to playback recorded real world events that were logged during real operational missions or tests. This allows recreation of faults that are otherwise extremely difficult to isolate and repair. Even so, it is almost impossible to recreate the exact actual environment. Little details such as clock rates that vary slightly over time can cause an interrupt to be handled at a different time during playback. Once the responses of the computer change, it becomes extremely difficult for the simulator to compensate and to adjust the playback data to correct for the altered behavior.

The degree of accuracy of a simulator directly effects its usefulness. It also usually effects the cost of the simulator. That is, simulators that do not accurately reflect the operation of the real system can do more damage than good, and simulators that very accurately reflect the real system tend to be very costly to build.

In summary, with the transportability of Ada, it is now sometimes easier to implement a simulator, since the operational code that would normally drive the real system can now be transported to a simulator with less effort in many cases. In this way, pieces of incomplete hardware can be substituted for by similar computers running the same code that the real hardware will run. This does not help solve the (perhaps more difficult) problem of real world environment simulation, but since the proliferation of microprocessors is so pervasive, many computers within weapons systems dedicate a large percentage of their control software to communications with other computers, and therefore a corresponding percentage of the software can be tested using a computer to simulate the communications.

Problem Resolution: A problem resolution is not required here.

See the general problem resolution under "Software Development Activities and Related Tools" above.

3.2.6.3 Automatic Regression Testing

Issue/Problem Definition: Automatic Regression Testing poses no new additional problems due to the Ada language. It is included here only for completeness.

Background: Regression testing is the selective retesting to detect faults introduced during modification of a system or a system component, to verify that modifications have not caused unintended adverse effects or to verify that a modified system or system component still meets its specified requirements. [11]

This information was supplied by the following sources: Current Software Engineering Literature, and LabTek Experience.

Analysis & Support: The process of regression testing often consumes a substantial portion of the effort for large applications that undergo many releases over their lifetime. In the absence of proper tools, regression testing is frequently incomplete and therefore it is not uncommon to have software regress. That is, in the release that fixes one error, other errors are introduced in functions that previously worked correctly. Most development environments do not have adequate facilities to automatically capture and re-run tests that were used to verify the initial system capability.

In summary, the transportability of Ada may provide the foundation upon which testing tools can be developed and provided to a wide group of contractors. The major difficulty is the intimate interfaces required to drive simulators in support of the testing. Since the simulators often operate in real-time the interfaces are often customized to a large degree. Provisions must be made within automatic testing tools to allow reconfiguration and customization of these interfaces.

Problem Resolution: See the general problem resolution under "Software Development Activities and Related Tools" above.

3.2.6.4 Correlation to Specified Test Verification Matrix

Issue/Problem Definition: No additional problems are imposed by the use of the Ada language with respect to correlation to the specified test verification matrix. It is included here only for completeness.

Background: A traceability matrix maps the software modules to the requirements. Each module, in turn, must have a test to show that it meets the requirements and a test verification matrix can be constructed. If a module is discovered which cannot be matched to a requirement, then that discrepancy should be resolved by eliminating the module or updating the matrix. [16]

This information was supplied by the following sources: Current Software Engineering Literature, and LabTek Experience.

Analysis & Support: Typically the verification matrix is not machine processable other than through a document formatter. A need exists to have the verification matrix be generated automatically from the raw input to the specifications, and to have a form of the matrix be used with the automatic test tools to insure that each test is verified. Use of specialized

comments within the raw (pre-formatted) text of specifications can identify the mandated requirements and also which test phases and how they will be verified.

In summary, the use of Ada, and standard software development guidelines (DoD-STD-2167A) should facilitate the development of tools to process the specification documents and provide the above capabilities.

Problem Resolution: See the general problem resolution under "Software Development Activities and Related Tools" above.

3.2.6.5 **Test Generation Assistance**

Issue/Problem Definition: There are no new problems associated with test generation assistance due to the Ada language. It is included here only for completeness. However, tools that generate test cases may need to be more complex to support the parsing of Ada programs unless they operate from an intermediate representation of the source Ada program.

Background: Test generation assistance is a software tool that accepts as input a computer program and and test criteria, generates test input data that meet these criteria, and, sometimes, determines the expected results. [11]

Testing within the context of software engineering is actually a series of four steps that are implemented sequentially. Initially, tests focus on each module individually, assuring that it functions properly as a unit, hence unit testing. Next, modules must be assembled or integrated to form the complete software package. Integration testing addresses the issues associated with the dual problems of verification and assembly. Finally, validation requirements (established during the planning phase) must be tested. Validation testing provides final assurance that software meets all functional and performance requirements. The last step falls out of the software engineering category and into the computer system engineering category. System testing verifies that all elements mesh properly and that overall system function and performance are achieved. [9]

This information was supplied by the following sources: Current Software Engineering Literature, and LabTek Experience.

Analysis & Support: Tools already exist within the Ada software repository which will instrument Ada source code and count the times each path is executed. Although this is not sufficient for many real time systems it does provide a baseline for verifying that each path has been executed. This can be a benefit to see if the test generation tool is providing sufficient test coverage to check all paths. Additional tools are required to assist in producing test cases.

In summary, this issue, like many other areas in software development is not unique to Ada, but may benefit from the common support of Ada for tool transportability.

Problem Resolution:See the general problem resolution under "Software Development Activities and Related Tools" above.

3.2.7 Maintenance

Issue/Problem Definition: There haven't been any serious problems identified in this area, largely due to the lack of longevity of Ada.

Background: Software maintenance is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment. [11]

This information was supplied by the following sources: Project Interviews, current literature, and LabTek Experience.

Analysis & Support: Where Ada code has been maintained for some period of time, three minor issues have arisen. The first is the conversion from subset compilers to validated compilers. In some cases features available in the subset compilers became unavailable in validated compilers. Also, interpretations of the language reference manual have changed slightly effecting the validity of some programs. These issues are believed to be very short term, and of no significant consequence. The last issues that has been discovered by most developers when working on projects with more than a few thousand lines of code, is that the Ada USE clause should not be specified for any packages other than those predefined in the language. The reason for this is because of the difficulty in tracing the origin of the object or type definitions. Without an intelligent editor, it is extremely hard to locate which package an object was declared in when many context (WITH) clauses are used.

Problem Resolution: The general problems associated with maintenance of software will benefit by the greater use of automatic tools. See the general problem resolution under "Software Development Activities and Related Tools" above.

3.3 Management & Policies

The following sections discuss the problems encountered by management in preparation of an Ada contract. Among the issues concerning management are: proposal development, resource allocation, software reuse policies, and training of personnel.

3.3.1 Proposal Development

Issue/Problem Definition: Sufficient time and analysis is not always spent during proposal development, especially regarding the impact of using Ada for the first time.

Background: The complexity and size of software systems, and their role in weapon systems has been increasing dramatically in recent years. Many contractors do not fully appreciate the difficulties associated with developing the software for these more complex systems. When little or no experience in using Ada exists within a company, the tendency is to assume that Ada is simply another language and that using it will have no impact on development.

This information was supplied by the following sources: Current Software Engineering Literature, and LabTek Experience.

Analysis & Support: Software is frequently given less analysis than the corresponding hardware components. Estimation of the development effort remains largely guesswork

and prone to underestimation. The impact of working towards impossible schedules that can not be changed until they become absolutely absurd is severe. Engineers are forced to start coding prior to understanding the requirements, let alone having a completed design. A tradeoff must be made between legislating the time it takes to develop a program and allowing the development to proceed indefinitely.

Problem Resolution: (Short Term) Contractors should justify their use of Ada and demonstrate that they understand the impact of using Ada and are prepared to support the transition to Ada.

(Short Term) The proposal phase should be expanded to include more effort on analyzing the software complexity and estimation of the corresponding development activity. The whole process of providing schedules needs to be studied.

3.3.2 Resource Allocation

Issue/Problem Definition: The resources for an Ada project are quite different from other software efforts and this must be taken into account.

Background: Resources for a software project include computer resources, software resources, and human resources.

This information was supplied by the following sources: Project Interviews, and LabTek Experience.

Analysis & Support: Additional hardware resources are required for development due to the mere size of the compilers and support tools. Due to the inter-dependencies of Ada program units, many recompilations impose a larger workload on the development system. In addition, extra human resources are needed to deal with problems associated with learning and using new tools.

Problem Resolution: (Short Term) The learning curve associated with a completely new, complex tool set must be taken into account during proposal and early planning phases.

(Short Term) Related to the proposal phase above, the planning of what personnel, computer and facility resources will be required is essential. Contractors should be given a "lead-time" that notifies them at least three months prior to when work is to begin, to prepare for the contract. Due to the uncertainties of government contracts, it often happens that in one month a contractor has an excess of software engineers, and in the next month they have a requirement for two hundred more. Even when it is possible to hire the people, preparing facilities for them is often a time consuming process, especially when security issues are involved.

(Short Term) Management within many DoD contractors needs to be advised of reports that indicate improved cost effectiveness of providing two-person office environments with terminals and work tables for each engineer. It is astonishing that contractors seem perfectly willing to pay consultants more than what the United States Secretary of Defense makes, and yet supply them with a small desk in a large crowded room full of ringing phones; then expect them to be able to design the most complex software system in the world.

3.3.3 Reusable Software

Issue/Problem Definition: Several difficulties arise in the reuse of software, including: locating the software for reuse, providing documentation that is compatible with the remainder of the system, legal and financial issues, and understanding, modifying and testing the software to operate in the new environment.

Background: Software reuse is the extent to which a module can be used in multiple applications. Once it is located, it must be understood. Building something out of parts that are not understood is difficult (and undesirable!). This is especially true for real-time software where the timing characteristics are determined by the analysis and understanding of a program's operation.

This information was supplied by the following sources: Project Interviews, Current Software Engineering Literature, and LabTek Experience.

Analysis & Support: To be reusable, a software routine needs to fit into the total system: that requires much more than just the program code. Documentation, specifications, design history, test plans and data, and all the other things required of the total system must be available for this component. In too many cases the work of finding and reusing is more than that of reinventing, but that leads to a world where everything is custom made and unique; something that cannot be afforded in today's large systems. [16]

Probably more difficult than the technical issues are the legal problems associated with software reuse. Who owns the software components? Who is responsible for their correct behavior? How should the owners be compensated? How can contractors be rewarded for the ultimate savings associated with software reuse?

Developing reusable software for real-time systems is particularly difficult since the optimizations often require taking advantage of the specific hardware. Also, contractors maximize profit when developing software in the most expedient fashion, rather than expending the extra effort and cost to design it for reuse. The recipients of reuse are most likely other contractors, and not necessarily the original developer.

Problem Resolution: (Short Term) Recent discussions with the STARS (Software Technology for Adaptable, Reliable Systems) office indicate there is an intention to upgrade and expand the software repository on SIMTEL20 (ARPANET). The majority of this software however, is related to tools and not real-time software.

(Short Term) Reusable software must be designed for reuse in mind. This implies a need to know what variances are allowed in different Ada implementations and and understanding of the concepts of Ada packaging. Also, a mechanism must be developed to measure the ability to reuse a particular piece of software before the reuse effort begins. This will provide a level of confidence that the reuse effort will be successful and will help to eliminate the concerns established by previous attempts at reuse, only to result in a wasted effort and a need for a total rewrite.

(Short Term) There is a need to provide a transportability handbook for Ada to assist engineers in this effort.

(Long Term) The government ıould establish a policy that rewards contractors for producing software that is reusable by other contractors. This is contrary to their natural tendencies and will require substantial wisdom to develop.

3.3.4 Training

Issue/Problem Definition: In addition to the usual Ada/Software Engineering training that is required, there is a general lack of knowledge about Ada runtime environments (RTEs) in the user community.

Background: Ada training takes on several forms ranging in effectiveness and cost. The forms of training are: on the job training, college level courses, self-paced video tape lectures and computer aided instruction, and concentrated two or three week courses given by professional instructors. It is generally held throughout the Ada community that courses teaching the language without hands-on programming assignments are of little value.

This information was supplied by the following sources: Project Interviews, Current Software Engineering Literature, and LabTek Experience.

Analysis & Support: Many of the projects interviewed had some training program, but they were generally less than three weeks. Untrained or improperly trained programmers tend to use Ada in one of two modes: either to mimic their most comfortable programming language (typically FORTRAN), or to go to the other extreme and use every possible feature of the language, often in places that are inappropriate.

Elegant Ada solutions, like tasking and generics often result in slower and larger programs. If sophisticated features of the Ada language are used when they are not necessary, the application will usually perform poorly. For example, tracking targets by creating an Ada task for each target, rather than using an array of records and a loop, will most likely give unsatisfactory results in a single processor application. There is a tendency by some to use the most "elegant" Ada solution rather than designing for real-time.

Interviews indicated that some problems with their designs did not appear until late in the development. In retrospect, they would have used other techniques to implement their application had they known the performance impacts of the chosen technique.

Problem Resolution: (Short Term) This problem can be corrected by proper Ada training. Careful attention must be given to make sure that current Ada implementation problems do not create a long term influence on the programming styles however. A separation must be made between features that are always likely to be less efficient than those that are inefficient because of current technology. Since Ada is new to many programs, the need for additional training is substantial. This goes beyond simple instruction in the syntax and semantics of Ada, but to the software engineering principles that are supported by the language. Managers and software selection groups must also be trained in Ada technology. For example, users are often mislead by status of validation. The AJPO seal of approval, or "validated" status of a compiler should not be interpreted to mean that the compiler is usable for all applications.

(Short Term) Proposals that include provisions for a large number of Ada software engineers should also show a capability to train their staff. This must include time in the schedule specifically set aside for training. A percentage (10%) of the staff should have

Ada experience on a previous project, to provide on-going support and training in Ada details that will not be absorbed during classes.

(Short Term) Contractors and Program Managers should obtain the AJPO's Catalog of Resources for Education of Ada Software Engineering (the CREASE) available through the Ada Information Clearinghouse (AdaIC).

4. Issue Vs. Source Matrix

The issues that were defined in the text are tied to their source in the matrix below. The first column contains the issue and the next four columns contain the sources. The sources are: interviews, ARTEWG & SIGAda meetings, current literature, and LabTek experience. An "X" is placed in the box to indicate the sources of each issue.

Source Issue (Text Section)	Interviews	ARTEWG & SIGAda Meetings	Literature	LabTek Experience
3. Software Engineering Issues Related to the Usa of Ada Technology In Real -Time Embedded Systems	X	X	X	X
3.1 Compilation Systems	X	X	X	X
3.1.1 Runtime Environments	X	X	X	X
3.1.1.1 Configurability	X	X	X	X
3.1.1.2 Execution Performance	X	X	X	X
3.1.1.3 Evaluation	X	X		
3.1.1.4 Size	X			X
3.1.1.5 Dynamic Priorities		X		X
3.1.1.6 Parallel Processing		X		X
3.1.1.7 Support of Low Level Operations	X			X
3.1.1.8 Task Restart		X		
3.1.1.9 Cyclic Scheduling		X		X

Ada Technology Issues

Issue (Text Section)	Interviews	ARTEWG & SIGAda Meetings	Literature	LabTek Experience
3.1.1.10 Coprocessor Support	X			X
3.1.1.11 Distributed Processing	X	X	X	X
3.1.1.12 Multilevel Security Support	X	X	X	
3.1.1.13 Reliability	X	X		X
3.1.2 Code Quality	X	X	X	X
3.1.3 Documentation	X			X
3.1.4 Validation	X		X	
3.1.5 Proposed Ada Language Extensions	X	X		X
3.1.5.1 Fast Interrupts	X	X		X
3.1.5.2 Greater Control of Task Control Block (TCB)	X			X
3.1.5.3 Asynchronous Task Signalling	X	X		X
3.1.6 Chapter 13	X	X	X	X
3.2 Software Development Activities and Related Tools	X	X	X	X
3.2.1 Requirements Analysis		X	X	X
3.2.1.1 Rapid Prototyping		X	X	X
3.2.1.2 Requirements Tracing		X	X	X

Issue (Text Section)	Interviews	ARTEWG & SIGAda Meetings	Literature	LabTek Experience
3.2.2 Configuration Management			X	X
3.2.3 Design	X			X
3.2.3.1 Flow Diagrams		X	X	X
3.2.3.2 Program Design Language (PDL)	X	X	X	X
3.2.4 Documentation		X	X	X
3.2.5 Implementation	X			X
3.2.6 Integration	X		X	X
3.2.6.1 Debuggers	X			X
3.2.6.2 Simulation	X			X
3.2.6.3 Regression Testing			X	X
3.2.6.4 Correlation to Specified Test Verification Matrix			X	X
3.2.6.5 Test Generation Assistance			X	X
3.2.7 Maintenance	X		X	X

Ada Technology Issues

Issue (Text Section)	Interviews	ARTEWG & SIGAda Meetings	Literature	LabTek Experience
3.3 Management & Policies	X		X	X
3.3.1 Proposal Development			X	X
3.3.2 Resource Allocation	X			X
3.3.3 Reusable Software	X		X	X
3.3.4 Training	X		X	X

5. Summary

Issues that impact the success of a software development in Ada can be classified into three primary categories: Compilation Systems, Software Development Activities & Related Tools, and Management and Policies. The area that is creating the most difficulties is compilation systems. Within this category, Ada Runtime Environments (RTEs) are the largest single impediment to using Ada in real-time embedded applications. Other areas of concern include: management understanding of Ada technology and its demand on resources, and the efficiency of the generated code.

For the current state of Ada technology there are two classes of real-time embedded applications which are not able to use Ada given the state of the current technology. The first class are the implementations with 8K bytes or less of memory capacity. Although the Ada runtime environment sizes are becoming smaller, they are not small enough yet. Even a customized version is too large for this class of applications. There is no benefit to forcing Ada into this memory size. It will require contortions that are likely to make the code less reliable, and the effort of rewriting these applications is unlikely to be a major cost given their small size. The second class of real-time embedded applications which should not be required to use Ada are the microprogrammed custom hardware systems. It is unlikely that Ada compilers will be produced for the large number of unique processors built from bit-slice components.

A common difficulty is selecting an Ada implementation, especially the runtime support routines, that will meet the needs of the application. The first source of information should come from the compiler vendors. The application requirements should be explained to the compiler vendor and the compiler vendor should explain how their runtime fits (or does not fit) the requirements. It may be necessary for the contractor to sign a non-disclosure agreement, since occasionally the details of the commercially available runtimes are proprietary. Obviously, the application engineers need to be included in the compiler selection process to evaluate the tradeoffs involved.

Although compiler vendors are becoming more knowledgeable regarding real-time software development, they still need more information about the nature of embedded applications and the relative priorities for addressing the problems associated with the RTEs. On the other hand, defense contractors understand the application requirements but they require education on Ada runtimes with respect to the Ada language features.

Ada can be used successfully on real-time embedded systems, even with traditional methods. The full benefits of Ada may not be fully realized however, if new methods are not adopted to facilitate issues such as maintainability and reusability.

Ada projects can run into serious problems at several different points in the life-cycle. Initially, in the design phase there may be serious training problems and a need to provide better guidelines for use of an Ada Program Design Language (PDL). The most likely "critical" area occurs in the test/debug phase where timing and sizing measurements may differ substantially from original predictions. A method for rapid prototyping should be utilized to help identify these problems earlier in the development phase.

6. Summary Of Interviews

LabTek prepared a list of questions that would provide much of the information needed. These questions were used as a guide for the discussions when interviewing each project manager, software manager, and application engineer. Not all questions could be answered by everyone and some clarification of the questions was needed. The following is the information received from the projects. The contributions of all the individuals who assisted with this effort is acknowledged and appreciated. The information that they supplied will assist in making positive changes to the way Ada is used in U.S. Army programs.

The following is a list of the most prominent issues reported:

- Runtime Library Too Large

The PM or contractor reported that the runtime library was too large (size in bytes) for the memory capacity of the system. In most cases, this was a problem if memory was a constraint.

- Required Customized Runtime Library

The PM or contractor reported that it was necessary to modify the runtime library. Customization was performed for a number of reasons, the main one being the runtime library was too large for the memory capacity. The features not absolutely needed were stripped out. The other reason that it was modified was to improve the performance.

- Generated Code Not Sufficiently Optimized

The PM or contractor reported problems due to the quality of the generated code. The generated code was not as efficient as code that could be produced by a programmer coding in assembler.

- Memory Was a System Constraint

Memory capacity was a constraint and therefore required careful management of it.

- Lack of Chapter 13 Posed a Problem

The PM or contractor reported problems due to the insufficient implementation of Chapter 13 of the Reference Manual for the Ada Programming Language. It is interesting to note that in Figure 3.,"Percentage Table for Reported Problems", this problem had the largest discrepancy between what the contractors reported and what the PMs reported.

- Tasking Algorithm Unusable For Real-Time

The PM or contractor reported that the tasking features were too time consuming to be used in a real-time application.

- Compiler Documentation Was Inadequate

The PM or contractor reported that the compiler documentation was inadequate for the problem at hand.

Figure 3., "Percentage Table for Reported Problems", provides the percentages of the interviews that reported the same problem. The left most column identifies the problems. The second column contains the percentage of PMs and contractors combined that reported the same problem. The numbers in this column are based on sixteen (16) different interviews. The third column contains the percentage of PMs only that reported a specific problem. The numbers in this column are based on interviewing eight (8) PMs. The fourth column contains the percentage of the contractors only that reported the same problem. It is based on interviewing eight (8) contractors.

Figure 3. Percentage Table for Reported Problems

REPORTED PROBLEM	PMs & CONTRACTORS COMBINED (16)	PMs ONLY (8)	CONTRACTORS ONLY (8)
RUNTIME LIBRARY TOO LARGE	69%	88%	50%
REQUIRED CUSTOMIZED RUNTIME LIBRARY	69%	63%	75%
GENERATED CODE NOT SUFFICIENTLY OPTIMIZED	81%	88%	75%
MEMORY WAS A SYSTEM CONSTRAINT	56%	50%	63%
LACK OF CHAPTER 13 POSED A PROBLEM	50%	13%	88%
TASKING ALGORITHM UNUSABLE FOR REAL-TIME	69%	63%	75%
COMPILER DOCUMENTATION WAS INADEQUATE	63%	50%	75%

7. Glossary

For definitions of standard software engineering terminology the reader is asked to refer to the ANSI/IEEE Std 729-1983, "IEEE Standard Glossary of Software Engineering Terminology". Terminology not defined in the IEEE standard will be defined in this glossary.

The **"Ada community"** is comprised of people in industry, academia, and the government using Ada technology for their specific requirements. It also includes the group of people who attend the SIGAda Conferences, and the readers of Ada LETTERS. The Ada Runtime Environment Working Group (ARTEWG) as well as the Ada compiler vendors are also part of the "Ada community".

"Ada Technology" consists of the currently available Ada compilation systems, tools, and associated methodologies.

An **"embedded system"** is a computer that is programmed to provide specific functions and integrated into a much larger system. Typically, the embedded computer is used to control or monitor the operation of the larger system. An example would be a pilot's advisory system onboard an aircraft. Its function is to provide the pilot with information such as aircraft roll, pitch, heading, weight, fuel reserves, etc., and it is integrated with the entire flight system.

A **"real-time"** system can best be differentiated by its quality of responsiveness. The question of how responsive a system must be before it merits designation as real-time is, of course, a relative one. For this report, "real-time" will mean that certain processing must take place in a time critical range to insure proper system functionality. That is, the system will not operate at all if this time is exceeded, as opposed to operating in a degraded mode.

It follows from the above two definitions that a **"real-time embedded system"** is a computer, programmed to perform a specific function, integrated into a much larger system that must respond to external events in an expedient manner to provide the functionality required. Characteristics of real-time embedded system software are the following:

- time critical calculations must be performed periodically (in a frame time, typically in the millisecond range)
- memory is usually limited
- interrupts signal an external event and must be handled in an expedient manner (typically in the *microsecond range*)
- some type of data transmission is to take place if it's a distributed system.

Examples of computations and functions performed by real-time embedded systems are:

- the sampling of inertial navigation data

* the driving of servos
* performing ballistic calculations
* providing data for a feedback loop
* transmitting/receiving data from one node of a distributed system to the next
* performing signal processing
* performing Kalman filtering.

"**Runtime Support (often called runtime)**" is the set of procedures or functions required to support the code generated from a compilation (i.e. entry call in a task, exception raising, abort processing, string catenation,etc.).

The "**Runtime Support Library**" (RSL) is the library of procedures and functions from which the runtime routines are selected. The runtime consists of a subset of the RSL.

The "**Runtime Environment**" (RTE) includes all of the runtime routines, the conventions between the runtime routines and the compiler, and the underlying virtual machine of the target computer. Virtual is used in the sense that it may be a machine with layered software (a host operating system). An RTE does not include the application itself, but includes everything the application can interact with (see Figure 2). Each layer has a protocol between it and the layer underneath it for interfacing. In the event that there isn't an operating system layer, the runtime includes those low-level functions found in an operating system.

"**Software engineering**" is the discipline and skillful use of suitable software development tools and methods as well as sound understanding of certain basic principles relating to software design and implementation.

The "**Software Engineering community**" is comprised of people in academia, industry, and government who are involved in the fields of computer science and software engineering. It also includes the literature such as: IEEE Transactions on Software Engineering, IEEE Software, ACM Software Engineering Notes, among other publications.

8. References

[1] ACM Ada Letters (continuing), including many bibliographies.

[2] ANSI/MIL-STD-1815A-1983. "Reference Manual for the Ada Programming Language", American National Standards Institute, Inc., 1983.

[3] Ichbiah, J.D., Barnes, J.G.P., Firth, R.J., Woodger, M., "Rationale for the Design of the Ada Programming Language", U. S. Government, AJPO, 1986.

[4] Ada Runtime Environment Working Group of SIGAda, "A White Paper on Ada Runtime Environment Research and Development", February 13, 1987.

[5] Ada Runtime Environment Working Group of SIGAda, "A Canonical Model and Taxonomy of Ada Runtime Environments", November 13, 1986.

[6] Jean E. Sammet, IBM Federal Systems Division, "Why Ada Is Not Just Another Programming Language", Communications of the ACM, vol. 29, no. 8, pp. 722-732, August 1986.

[7] Ware Myers, "Ada: First users - pleased; prospective users - still hesitant", Computer, pp. 68-73, March 1987.

[8] Rzepka, William and Ohno, Yutaka, "Requirements Engineering Environments: Software Tools for Modeling User Needs', COMPUTER, pp. 9-12, April 1985.

[9] Pressman, Roger S., "Software Engineering: A Practitioner's Approach", McGraw-Hill, 1982.

[10] Carrio, Miguel, "Life Cycle and Ada", Defense Science & Electronics, pp. 17-24, July 1986.

[11] IEEE Std 729-1983, "IEEE Standard Glossary of Software Engineering Terminology"

[12] Gannon, J.D., Katz, E.E., Basili, V.R., "Metrics for Ada Packages: An Initial Study", Communications of the ACM, Volume 29, No. 7, pp. 616-623, July 1986.

[13] Gomaa, Hassan, "Software Development of Real-Time Systems", Communications of the ACM, Vol. 29, No. 7, pp. 657-668, July 1986.

[14] Fisher, D.A., Weatherly, R.M., "Issues in the Design of a Distributed Operating System for Ada", COMPUTER, pp. 38-47, May 1986.

[15] Spiegel, M., "Software Prototyping", Colloquium Series, Wang Institute of Graduate Studies, March 1981.

[16] Mathis, R. F., "The Last 10 Percent", IEEE Transactions on Software Engineering, vol. se-12, no. 6, pp. 705-712, June 1986.

[17] Kennedy, T., "Advances in smart munitions", Defense Science & Electronics, pp. 63-67, October 1986.

[18] "TRW Multi-level Secure Tactical Operating System", Defense Science & Electronics, pp. 68, January 1987.

[19] "VADS VERDIX Ada Development System, SUN-3/UNIX, VADS Version 5.41", Verdix Corporation, 1986.

[20] "VERDIX Ada Development System - VADS Version 5.41 for SUN-3/UNIX => Motorola 68000 Family Processors", Verdix Corporation, 1987.

FINAL TECHNICAL REPORT

SOFTWARE ENGINEERING PROBLEMS USING ADA IN COMPUTERS INTEGRAL TO WEAPONS SYSTEMS

by

Sonicraft, Inc.
8859 S. Greenwood Avenue
Chicago, IL 60619

for

U.S. Army HQ CECOM
Center for Software Engineering
Advanced Software Technology
Fort Monmouth, NJ 07703-5000

9 OCTOBER 1987

TABLE OF CONTENTS

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Purpose

It is now the policy of the U.S. Department of Defense (DoD) that Ada shall be the single, common, high-order programming language for all computers that are integral to weapon systems [Ref. 1]. Use of validated Ada compilers is required on all projects, as is the use of Ada-based program design language (PDL).

This technical report investigates some of the generic problems that have arisen when this policy has been followed on actual projects and relates them to the software engineering principles embodied in DoD Standard 2167 [Ref. 2].

1.2 Terminology

The definitions given in 2167 apply to all terms used in this report, with additional terms defined in ANSI/IEEE Std 729-1983 [Ref. 3] and in Paragraph 3 of this report.

1.3 Organization

The report is organized to first show the methods used to obtain the reported problems in Paragraph 4. The problems themselves are then analyzed according to the categories of software engineering principles which they affect, with the results of the analyses collected in dBASE III compatible files. These are available in computer-readable format as well as being presented in the Appendix.

The main findings of this report are summarized in Paragraph 5, which contains a list of the problems, the relative importance of each problem, and the categorization of each problem in the analysis matrix used by the data base. Paragraph 5 also contains the Sonicraft recommendations for the use of the information presented herein.

2. REFERENCES

[1] Department of Defense Directive 3405.2, SUBJECT: Use of Ada in Weapon Systems, March 30, 1987

[2] DOD-STD-2167, Defense System Software Development, 4 June 1985

[3] ANSI/IEEE Std 729-1983, IEEE Standard Glossary of Software Engineering Terms

[4] W. Myers, "A Statistical Approach to Scheduling Software Development," IEEE Computer, Dec 78

[5] G. M. Barnes, "Assessing Software Maintainability," Communications of the ACM, Jan 84

[6] F.P. Brooks, "No Silver Bullet," IEEE Computer, Apr 87

[7] F.P. Brooks, " The Mythical Man-Month, " 1975, Addison-Wesley, Reading, Mass.

[8] S.E. Watson, "Ada Modules," ACM Ada Letters, vii. 4-79, 1987

[9] C. Kemerer, "An Empirical Validation of Software Cost Estimating Models," Communications of the ACM, May 87

[10] B. Boehm, "Software Engineering Economics," Prentice-Hall, Englewood Cliffs, NJ, 1981

[11] H. Davis, "Measuring the Programmer's Productivity,"Engineering Manager, Feb 85

[12] R. W. Jensen, "Projected Productivity Impact of Near-Term Ada Use in Software System Development, "Hughes Aircraft Co., Fullerton, CA, 1985

[13] S. Boyd, "Ada Methods: Object-Oriented Design & PAMELA, SIGAda, Nov 86

[14] MGEN Smith, Policy Committee Reports From Armed Services, SIGAda, Nov 86

[15] MGEN Salisbury, Policy Committee Reports From Armed Services, SIGAda, Nov 86

[16] S.J. Hanson and R.R. Rosinski, "Programmer Perceptions of Productivity and Programming Tools, " Communications of

Productivity and Programming Tools, " Communications of the ACM, Feb 85

[17] G. Booch, " Software Engineering with Ada, " 1983, Benjamin/ Cummings.

[18] Labtek Corporation, Subject: Software Engineering Issues On Ada Technology Insertion For Real-Time Embedded Systems, 24 July 1987.

3. DEFINITIONS

Runtime Support or Runtime Support Library (RSL): The set of embedded firmware required to interface the applications object code generated by the Ada compiler to the target machine instruction set.

MEECN: Minimum Essential Emergency Communication Network, a stand-alone network that is intended to continue operating reliably after the primary (higher bandwidth) communication network has failed (such as after a natural disaster or a nuclear event).

PAMELA: Process Abstraction Method for Embedded Large Applications, a trademark of George Cherry for his Ada design methodology.

Real-Time Function: Any system function (hardware, software or a combination) which is considered to have faulted if it has not been completed within a specified time after a signal to start.

Real-Time Ada: A computer program written in the Ada language which implements one or more real-time functions, usually triggered by interrupts.

Efficiency: Efficiency deals with utilization of resources. The extent to which a component fulfills its purpose using a minimum of computing resources. Of course, many of the ways of coding efficiently are not necessarily efficient in the sense of being cost effective, since transportability, maintainability, etc., may be degraded as a result.

Integrity: Integrity deals with software failures due to unauthorized access. and is the extent to which access to a component or associated data by unauthorized persons can be controlled.

Reliability - Those characteristics of software which provide for definition and implementation of functions in the most non-complex and understandable manner. By reducing complexity the chances of the program providing the functions as intended are increased, thereby improving reliability.

Survivability - Survivability deals with the continued performance of software (e.g., in a degraded mode) even when a portion of the system has failed.

Usability - User effort required to learn, operate, prepare input for, and interpret output of a component.

Correctness - The extent to which software design and implementation conform to specifications and standards.

Criteria of correctness deal exclusively with design and documentation formats, such as agreements between a component's total response and the stated response in the functional specification (functional correctness), and/or between the component as coded and the programming specification (algorithmic correctness).

Maintainability - The ease of effort in locating and fixing software failures. The extent to which a component facilitates updating to satisfy new requirements or to correct deficiencies. This implies that the component is understandable, testable, and modifiable.

Verifiability - Software design characteristics affecting the effort required to verify software correctness.

Portability - A property of a program representing ease of movement among distinct solution environments.

Reusability - The extent to which a component can be adapted for use in another environment (e.g., different host or target hardware, operating system, APSE) or another application.

Expandability (Extendibility, Flexibility) - A measure of the effort in increasing software capabilities or performance or to accommodate changes in requirements, or the extent to which a component allows new capabilities to be easily tailored to user needs.

Training -Those characteristics of software which provide transition from current operation and provide initial familiarization. A measure of the extent to which training and other user help is available from the vendor of a component or from the component itself, including on-line, documentation, listings, and printouts, which serve the purpose of providing operating instructions for using the component to obtain desired results.

Configuration Mgmt - A measure of the discipline related to controlling the contents of a component, including monitoring the status, preserving the integrity of released and developed versions, and controlling the effects of changes throughout the component.

Costs - The cost of a complete component or the costs of features of a component. Other cost considerations are installation, user assistance, and maintenance support.

Allocator - An allocator creates a new object of an access type, and returns an access value designating the created object.

Elaboration - Elaboration is the process by which a declaration achieves its effect. For example it can associate

a name with a program entity or initialize a newly declared variable.

Exception - An exception is an event that causes suspension of normal program execution. Bringing an exception to attention is called raising the exception. An exception handler is a piece of program text specifying a response to the exception. Execution of such program text is called handling the exception.

Rendezvous - A rendezvous is the interaction that occurs between two parallel tasks when one task has called an entry of the other task, and a corresponding accept statement is being executed by the other task on behalf of the calling task.

Task - A task is a program unit that may operate in parallel with other program units. A task specification establishes the name of the task and the names and parameters of its entries; a task body defines its execution. A task type is a specification that permits the subsequent declaration of any number of similar tasks.

Visibility - At a given point in a program text, the declaration of an entity with a certain identifier is said to be visible if the entity is an acceptable meaning for an occurrence at that point of the identifier.

4. APPROACH

Sonicraft used a five step approach to meet the objectives of this program:

1) Identify the sources which could identify problems in developing Ada software for computer which are integral to weapon systems.

2) Identify the relevant problems.

3) Isolate the problems that are generic.

4) Classify the problems based on which aspects of Software Engineering are affected by each problem.

5) Analyze the effects of each problem for each of these categories.

4.1 Sources

The main source of problems was our own experience in developing the MEECN system, which is the first weapon system developed using the Intel 8086 microprocessor and the full Ada programming language (including tasking and other non-Pascal features). There were literally hundreds of problems which were encountered in this development, and most were at least thought to be Ada-related at the time of their occurrence.

Other sources which were planned included contacts from the Ada Joint Program Office (AJPO) Evaluation and Validation (E&V) Team; other personal contacts from the government, industry and academia; information from SIGAda, the National Ada Symposium, and other Ada conferences; and information from Ada-related publications.

Due to replanning to stretch out the period of performance of our task, some of the planned contacts were postponed to the next phase of this contract. These contacts should be made to broaden the data base of this study and to avoid the impression that it is based solely on the experiences of one organization.

4.2 Relevant Problems

Sonicraft capitalized on the MEECN project experience by concentrating on the Ada problems that arose from Intel targets. Since the MEECN project is still in formal test, and more hardware integration problems will probably still be found, Sonicraft also concentrated on the early phases of the Software Life Cycle.

Problems were accepted for this study only when the root cause had been determined unambiguously. In practice this meant that the problem had been solved and had not returned in subsequent software tests. This caveat was necessary because Sonicraft had found problems that appeared to be Runtime Support Library (RSL) problems of the Ada compiler, but were later found to be due to other causes. For example, the Intel Microprocessor Development System once had a probe which gave erroneous readings due to a calibration problem. Until this was discovered during a system verification by Intel, Sonicraft was searching for a nonexistent RSL problem.

4.3 General Problem

To be considered generic, a problem must have been reported by more than one source. In addition, problems that stemmed from a common cause were grouped together as symptoms of the root problem. As an example, Sonicraft experienced problems in using several Ada Language System (ALS) tools, but these were grouped together as part of a single problem (#16) rather than treating each separately. By doing this the original set of hundreds of problems was distilled to a more manageable (and understandable) set of.

By treating only generic problems Sonicraft expects that users of this report will be more likely to find the problems they can expect to encounter.

4.4 Classification

The generic problem impacts were classified into six major categories:

1) Attributes of the delivered application

2) Methodology used to develop the application

3) Implementation of the application

4) Tools used to develop the application

5) Phase of the Software Development Cycle where the problem occurs

6) Management impacts on the application

These categories were then subclassified into 51 subcategories, most of which were found to be applicable to at least one Ada problem. This part of the process is very amenable to update, as new or better categories can be added reasonably quickly with the more manageable set of 37 problems to work with.

4.5 Analysis

The analysis of each generic problem consists of first a problem definition, then detailed write-ups showing its impact on each applicable Software Engineering subcategory. The detailed analysis establishes exactly WHAT the problem is, references WHO (or WHERE) it came from, and explains WHY it is a problem. Its purpose is to provide enough background material so that the subcategory analyses will be more meaningful. This section normally concludes with a brief summary of Sonicraft's conclusions about the overall effect of the problem.

The detailed write-ups in each subcategory provide information needed for a full understanding of the analysis, such as WHEN the problem occurs in the Software Development Lifecycle, and how its symptoms (often unrecognized as rooted in this generic problem) may extend to later phases of the Lifecycle.

Where problems are related to other generic problems, the relationship is highlighted in the portions of the analysis where it is most needed to understand the problem under analysis.

Assumptions were not intentionally made in the analysis of any of the problems, but in several instances the opinion of a problem source is reported. These instances include specific references to the source so that the reader can investigate these opinions in more depth if desired.

In many cases a rationale, such as a developer of the Ada software for computers integral to weapon systems might use, is presented to help in understanding why something might be considered a problem. The use of a rationale is not a scientifically valid method to support a conclusion, of course, but was included where it helped to visualize the kinds of things that experienced developers worry about.

Notably missing from this analysis is the recommendation of what to do about the problems. This omission was deliberate because it requires additional data collection and analysis to arrive at sound conclusions. Even then these conclusions must be tested to achieve enough credibility to convince people to take the trouble to implement them. These efforts are beyond the scope of the present Sonicraft task.

5. SUMMARY AND RECOMMENDATIONS

5.1 Format of Problem Analysis

Figure 5-1 presents a list of the generic problems that were found to result from the use of Ada in computers that are integral to weapons systems. The importance shown for each problem is determined as follows:

VI = Very Important problem in terms of its impact on developers trying to use Ada. Requires awareness of the developers and needs attention to develop good solutions as soon as possible.

I = Important problem, but not as potentially damaging to a development effort as the VI problems.

LI = Least Important problem compared to the others, but still worthy of attention to develop a good solution.

Figure 5-1 also presents a matrix of these problems versus the software engineering categories they affect. The Appendix presents a definition of each problem, a definition of each category, and a problem impact analysis on each category wherever an "X" appears in the figure.

5.2 Summary of Findings

The generic problems seem to convey two messages over and over:

1) It is a lot more expensive to do programming in Ada.

2) Worse yet, for some applications it is not yet technically feasible.

Our conclusion after analyzing these problems is that situation is not necessarily that bleak for a developer starting out in Ada today. Some of the damage these problems can cause can be averted simply by knowing about them and then exercising reasonable vigilance (i.e., Problem #17). To the extent that this is possible, this report is intended to be of immediate benefit.

Another point to consider is that most of the Very Important problems are related to Ada compiler technology in some way. The focus of this report has been on cross-compilers targeted to the Intel 8086 microprocessor, which, while being the most widely-used microprocessor in the world, is not particularly well suited for Ada tasking. Subsequent products from Intel are much better in this regard. Also, the 8086-targeted compilers have markedly improved about every six months since the first one was validated less than two years ago. While it is dangerous to extrapolate this trend to the

	TITLE		ATTRIBUTES												METHODOLOGY					
			Efficiency	Integrity	Reliability	Maintenance	Correctness	Verify	Portable	Security	Complexity	Reusability	Expandability	Risk	Standards	Training	Tool Use	Reviews	Method Type	Documentation
1.	Lack of Knowledge Concerning Ada RTE	VI				X	X	X					X							
2.	Impact of Ada Comp Impl Differences	I	X			X														
3.	Impact of Intrpt Hndlg O/H	VI	X		X				X											
4.	Impact of Mem Mgmt O/H	VI	X				X	X		X										
5.	Impact of RSL O/H	VI	X																	
6.	Impact of Tasking O/H	VI	X																	
7.	Ineff of Obj Code Gen	VI	X																	
8.	Rqmts for Extensive Ada Optim	VI			X	X		X	X		X	X								
9.	Inadeq Cap Provided by Curr Debug	VI						X												
10.	Delayed Handling of Ada Exceptions	LI						X												
11.	Impact of Extensive Use of Generics	LI	X																	
12.	Inability to Perf Indep of RSL	I	X																	
13.	Lack of a Distr RSL (Multiple Proc)	VI	X	X																
14.	Inability to Perf Secure Ada Proc	VI		X						X	X									
15.	Diversity in Impl of APSE's	LI			X	X			X							X	X			
16.	Poor Performance of Ada Tools	I	X		X												X			
17.	Diff in Benchmarking Ada Systems	VI	X			X							X				X			
18.	Lack of Ada S/W Development Tools	I			X				X		X	X								
19.	Ada Language Complexity	LI		X				X			X									
20.	Rqmts for Customization of RSL	VI				X			X			X								
21.	Lack of Exp Ada Programmers	VI	X											X		X	X	X	X	
22.	Extensive Ada Trng Requirements	I														X				
23.	Inaccy of C/S Est for Ada Prog	VI												X						
24.	Lack of Est Ada SW Dev Method	VI																	X	
25.	Lack of Est Ada SW Stds & Guide	I																		
26.	Productivity Impacts of Ada	I																		
27.	Impact of Constraint Chk On Sys. Perf	I	X																	
28.	Inability To Assgn Dyn Tsk Priorities	I									X		X							
29.	Inability To Perf Parallel Processing	I	X								X		X							
30.	Lack Of Suppt For Low Level Operation	I			X		X						X							
31.	Inability To Perf Task Restart	I	X		X		X													
32.	Inability To Perf Cyclic Sch In Ada	VI					X						X							
33.	Lack Of Fltng Pt Coprocessor Support	I	X				X													
34.	Inability To Recover From CPU Faults	I			X		X													
35.	Impact Of Ada ACVC Issues	VI	X								X									
36.	Inability To Perf Async. Task	I	X										X							
37.	Lack Of Implementation Of Chapter 13	VI				X							X							

	IMPLEMENTATION										ADA TOOLS								
	Benchmarking	Simulation	Optimization	System Design	System Timing	Dev Environment	PDL Use	Non Ada Software	Vendor Inter	Prototype	Compiler	Linker	Importer	Exporter	Debugger	Editor	Config Manager	Prog Analyzer	PDL Processor
1.																			
2.	X								X	X					X				
3.	X	X	X	X	X					X								X	
4.	X	X	X	X	X				X	X	X							X	
5.	X	X	X	X	X					X	X							X	
6.	X	X	X	X	X					X								X	
7.	X	X	X	X	X													X	
8.						X		X	X		X								
9.															X				
10.	X																		
11.	X		X	X						X									
12.									X				X						
13.				X															
14.						X			X	X	X								
15.						X			X		X								
16.	X		X			X			X										
17.				X							X								X
18.						X													
19.																			
20.	X																		
21.	X		X	X	X		X	X	X	X									
22.																			
23.																			
24.																			
25.							X	X											
26.											X				X				X
27.				X	X														
28.																			
29.																			
30.																			
31.																			
32.																			
33.																			
34.																			
35.																			
36.																			
37.								X											

	S/W DEV CYCLE							PROJ. MANAGEMENT						
	Software Reqmt	Preliminary Des	Detailed Des	Code Test	CSC Test	CPCI Test	System Test	Personnel Reso	Facilities	Cost	Schedule	Estimation	Customer Rel	[illegible]
1.	X	X	X		X	X	X	X			X			
2.														
3.	X	X	X					X		X	X			
4.	X	X	X											
5.	X	X	X											
6.	X	X	X											
7.														
8.		X	X		X	X		X	X	X	X	X		
9.														
10.														
11.										X				
12.		X												
13.														
14.		X	X					X		X				
15.									X					X
16.										X	X			
17.	X	X	X	X					X	X	X	X		
18.										X	X			
19.			X					X						
20.				X										
21.								X		X	X		X	X
22.				X						X	X			
23.								X	X				X	X
24.								X	X	X	X	X	X	X
25.								X		X	X	X		
26.								X	X	X	X	X		
27.														
28.														
29.														
30.														
31.														
32.														
33.														
34.														
35.														X
36.														
37.														

FIG. 5-1 GENERIC PROBLEM MATRIX

future, it is nevertheless encouraging.

Finally, the tone of this report has been set by intentionally concentrating on the problems in the use of Ada. A similar report concentrating on thirty-seven advantages in the use of Ada for computers integral to weapons system could have been written instead, and it would have given the opposite impression. In fact, we see the introduction of Ada as a tremendous boost to productivity, but only after problems similar to those analyzed here have been addressed.

5.3 Recommendations

Sonicraft recommends that the poroblem data base be enriched in three ways:

1) Periodically reassess the problems since Ada technology, especially for compilers, is very dynamic.

2) Determine the effects of using current development methodologies on the frequency of problem occurrence.

3) Determine a problem avoidance matrix, constructured similarly to figure 5-1, showing what developers do to avoid or minimize the occurrence of generic problems.

These extensions and updates will magnify the benefits of this report and help speed the acceptance of Ada as a viable language for embedded processors.

APPENDIX 10
REAL TIME ADA PROBLEMS

10.1 LACK OF KNOWLEDGE CONCERNING THE ADA RUN-TIME ENVIRONMENT

10.1.1 Definition

The language provides a very complex, powerful, and sophisticated run-time environment to support such Ada features as memory management, process control, and others. The primary element in the Ada-supplied run-time environment is the Run-Time Support Library (RSL). The RSL performs a number of activities which include:

* Memory Management
  Dynamic memory allocation/deallocation
  Garbage collection

* Process Scheduling
  Task activation/deactivation
  Task scheduling
  Task rendezvous

* Resource Control
  Resource scheduling
  Resource monitoring

* Error Processing (exception handling)

The RSL resides in the target environment during system operation and operates in conjunction with the applications software.

The performance of the RSL affects the performance of the applications programs (tasking, memory management, interrupts, etc.). The RSL is also part of the system debug/testing process since it resides in the target environment. The RSL may require customization as part of the system optimization process. The RSL may need to be benchmarked to obtain an accurate estimate of system performance.

The information concerning the RSL characteristics is vital to the applications developers, since they must address the impact of the RSL during system design and implementation. If this information is not made available to the applications programmers, there is little hope that the system being developed will perform as expected. To date, the Ada compiler vendors have provided little information to the Ada applications developers concerning the detailed sizing, timing, performance, and functional characteristics of the RSL.

10.1.2 Maintenance

The implementation of changes to an Ada program is dependent on the capabilities of the RSL and the interfaces between the RSL and the programs being maintained. Lack of knowledge concerning the RSL could result in the implementation of program changes without a full understanding of their impact on system performance.

10.1.3 Correctness

The Ada run-time environment includes the resident RSL programs which interact with the applications program(s). Lack of knowledge concerning the RSL could lead to misunderstandings concerning the functions of the RSL and the applications programs. These programs could lead to the development of applications programs that do not perform as expected.

10.1.4 Verifiability

During the testing of Ada programs, errors that are found must be isolated to program components so that they can be corrected. Lack of knowledge concerning the RSL causes added difficulty in isolating and correcting these errors because it is difficult to determine whether the error is due to an incorrect applications program, a problem in the RSL, or a misunderstanding concerning the operation of the RSL or its interfaces to the applications programs.

10.1.5 Risk

The development problems that arise from a lack of knowledge of the RSL can greatly increase the risk associated with the development of an Ada applications program. Technical risk can result because applications programs might be developed that do not provide expected performance or capabilities due to an incomplete understanding of the efficiency and functional capabilities f the RSL. Extensive rework and redesign may be required to rework these deficiencies, because the changes that are made to correct these problems may impact other areas of the sytem. Cost and schedule risk are also factors and will be addressed as project management issues.

10.1.6 Software Requirements Analysis

Lack of knowledge of the RSL capabilities could prevent Ada software developers from performing an accurate software requirements analysis with regard to issues such as sizing, timing, performance, and functionality. Since the RSL operates in conjunction with the applications software, it is critical for Ada developers to know the characteristics of the RSL when attempting to determine the feasibility of meeting proposed system requirements.

10.1.7 Preliminary Design

Lack of knowledge concerning RSL functionality, performance, and interfaces could prevent Ada software developers from specifying correct and efficient interfaces between the RSL and the applications software. The performance of the RSL must be addressed while performing preliminary system sizing and timing analysis. Also, the operations that are performed by the RSL (task scheduling, memory management, etc.) must be considered when attempting to determine the various system states.

10.1.8 Detail Design

Lack of knowledge concerning the implementation details of the RSL could cause problems during the Detailed Design Phase. The run-time performance of the RSL must be addressed when performing the detailed system sizing and timing analyses. Accurate information concerning the interfaces between the RSL and the applications program is necessary to develop the detailed software interface definitions.

10.1.9 CSC Test

During CSC integration and testing, lack of knowledge concerning the functional, performance, and interface characteristics of the RSL makes it very difficult for the Ada developers to isolate software errors. Also, any problems that originate in the RSL itself (due to the immaturity of the run-time programs) are almost impossible to debug unless someone with an extensive knowledge of the RSL code (most likely supplied by the compiler developer) is available to the development team.

10.1.10 CSC Test

The problems that occur during this phase, due to lack of knowledge of the RSL, are similar to those described in the CSC testing phase. They all result from rework from earlier phases of tasks done in error due to RSL misunderstanding.

10.1.11 System Test

The problems that occur during the system testing phase due to lack of knowledge of the RSL are similar to those that occur in the CSC and CSCI testing phases. They are all rework of tasks in earlier phases, but the rework is much more expensive in this phase because of the formal testing that must be repeated.

10.1.12 Personnel Resources

Lack of knowledge of the RSL can result in incorrect design, which then requires sometimes extensive rework by senior staff members. This effort, along with the effort necessary to learn

about the characteristics of the RSL, requires the use of additional personnel resources.

10.1.13 Cost

The development problems that arise from a lack of knowledge of the RSL can adversely impact system development costs. This lack of knowledge can result in failure to design/implement correct interfaces between the applications programs and the RSL. It can also result in applications programs that do not provide expected performance or capabilities due to an incomplete understanding of the efficiency and functional capabilities of the RSL. The rework and, in some cases these problems can cause a significant increase in project development cost.

10.1.14 Schedule

The problems that can arise from having a lack of knowledge concerning the RSL characteristics could adversely impact the system development schedule.

10.2 IMPACT OF ADA COMPILER IMPLEMENTATION DIFFERENCES

The differences in Ada compiler implementation can impact the performance of an Ada application. The ACVC tests that are used by the AJPO to validate candidate Ada compilers primarily verify the functional and syntactical aspects of the compilers, they do not address performance and implementation issues. Between this and the optional implementation issues discussed in Chapter 13 of the Ada Language Reference Manual, a compiler vendor has some flexibility in determining a compiler implementation approach. The areas in which the impacts of compiler implementation differences are most likely to be observed are:

* The efficiency of the generated Ada object code
* The size of the RSL
* The run-time overhead associated with the RSL
* Implementation of language features (tasking, generics, etc.)
* Compilation speed (lines of code per minute)

Differences in compiler implementation can have an effect on system performance. They can cause a reduction in system efficiency, thus requiring additional system resources (hardware and software). Impacts can also be felt in the areas of system maintainability (for both the compiler and the applications software) and in modifications that must be made to other Ada support tools (such as the debugger) to reflect the compiler implementation differences.

10.2.1 Efficiency

The ACVC tests that are performed to validate an Ada compiler are primarily concerned with functional and syntactical issues; they do not address implementation and efficiency issues. Thus a different compiler implementation that provided reduced efficiency would still be validated if it passed the ACVC tests.

The performance of an Ada compiler is affected by a number of issues such as the algorithms employed by the compiler to perform its processing, the effectiveness with which the compiler-generated code utilizes the system architecture, and the performance (execution speed and memory usage) of the compiler-generated code.

If a different implementation of an Ada compiler did not perform as well in some of these system-specific and application-specific areas, the result could be a reduction in

the overall efficiency of the system.

10.2.2 Maintenance

A different implementation of a compiler could provide reduced maintainability for the compiler itself, if the implementation resulted in an increase in compiler complexity. The maintainability of the applications programs could also be adversely affected if the different compiler implementation caused an increase in the complexity of the compiler-generated object code.

10.2.3 Benchmark

To determine the anticipated impact, if any, of a different compiler implementation on system performance, the compiler should be benchmarked to assess its performance. Along with an evaluation of overall compiler performance, this benchmarking should address those areas of the compiler in which the different implementation could adversely impact system performance.

10.2.4 Vendor Interface

The differences in Ada compiler implementations could adversely impact performance of the compiler and the compiler-generated code. To obtain detailed and accurate information describing the compiler implementation and its impact on system performance, it is important to establish a relationship with the compiler vendor. The compiler vendor can provide this information to the applications developers and can help them relate these results to the system development.

10.2.5 Prototype

To determine the impact of the different Ada compiler implementations on system performance, a prototype of the the system should be developed. This allows the applications developers to obtain an actual evaluation of the impact of the different compiler implementations in their application environment.

10.2.6 Debugger

Differences in the implementation of the Ada compiler can impact the design of the system debugger. For example, differences in the target machine memory layout constructed by the compiler must be incorporated into the debugger so that it reflects the differences.

10.3 IMPACT OF INTERRUPT HANDLING OVERHEAD ON SYSTEM PERFORMANCE

With the embedded system being the primary target for the use of the Ada language, the efficient handling of interrupts becomes a major issue. Interrupts can be defined as hardware or software signals that stop the current processes of the system under specified conditions and in such a way that the processes can be resumed. In the embedded system environment, interrupts are critical to the ability of the system to respond to real-time events and perform its required functions. Interrupts, in general, signal the occurrence of some predefined event to the embedded system. The embedded system must then perform the correct functions in some determined amount of time. Therefore, any overhead time, time spent on processes that are over and above the required processes, will degrade the ability of the embedded system to meet its functional requirements. To better illustrate the problem, please note the following:

```
|<---------- B ------------->|<------------- C ------------->|
     A
```

A : Represents the occurrence of a interrupt.

B : Represents the overhead time required to stop the current process and switch to the interrupt handler process.

C : Represents the time spent performing the required processes in response to the interrupt.

The problem that exists with Ada today is the overhead time "B" that is required in a Ada program to stop and switch to the interrupt handling process is too large and in many cases unacceptable for embedded system applications. Currently, Ada programs have overhead times in the hundreds of microseconds to milliseconds. The non-Ada embedded system application overhead times have been in tens of microseconds or less.

10.3.1 Efficiency

The efficiency of the application program will be directly affected by the overhead of interrupt handling. Thus the system will inefficiently process the incoming interrupts. The more interrupts the system has to process per unit time will determine how inefficient the application program and the system will be.

10.3.2 Reliability

With the amount of overhead associated with the processing of embedded system interrupts, one must be very careful in the design of an Ada system. Because the period of time it takes to begin executing the interrupt handler can vary, it is possible for the system to occasionally miss an interrupt. This usually compromises the functioning of the system and may even shut it down.

10.3.3 Portability

Portability of Ada programs will prove to be more difficult, since interrupt processing in an implementation depends on features of the RSL. The timing involved in the execution of interrupts using one compiler cannot be guaranteed to work on another compiler.

10.3.4 Benchmark

Benchmarking is often required when a new set of tools is being evaluated for use on a given project. Because of the critical nature of interrupts to the performance of real-time embedded systems, there is a strong need to develop some detailed benchmark tests to evaluate the characteristics of the tools with respect to interrupts. By developing a set benchmark tests the project will be able to evaluate the various tools available and select the set of tools best suited to meet the requirements of the given project. This can produce major cost savings later in the life cycle because of proper tool selection.

10.3.5 Simulation

After the interrupt characteristics of a particular set of tools have been tested and the results show that there may be some risk of meeting the requirements, than a series of risk reduction simulations are required. The purpose of the simulation effort will be one to determine the impact on the design of the system and two what course of action must be taken for risk reduction. To perform these steps the simulation effort will simulate a subset of the required functions of the embedded systems. The functions to be simulated are those functions that show a high degree of risk. For this particular problem of interrupt overhead impact, the designer's aim is to reasonably prove or disprove that he can meet the requirements of the project. Simulations are one of the best ways to determine early in the project life cycle the feasibility of a design approach.

10.3.6 Optimization

Once simulations have been developed and the designer can experiment with the problem of interrupt overhead impacts, he can determine if there is any need to optimize either the tools or his design approach. The impact of performing optimization can be delays in project schedules, cost over-runs, and loss of readability and maintainability. Thus, optimiza-tion effort would only be conducted if it is determined that it is required to meet the requirement of the project.

10.3.7 System Sizing

In the embedded system environment sizing is a critical factor and any overhead impacts will directly effect the sizing of the system.

Such is the case for the overhead associated with the use of Ada interrupts. Of course, benchmark testing and simulations should be conducted to determine the full impact of the overhead on the system sizing requirements. If it is determined that there is to much risk of not meeting the requirements, then corrective action, such as optimizing can be taken.

10.3.8 System Timing

Another critical factor in the development of the embedded system has to do with timing or the ability of the system to respond to events. Because embedded systems rely heavily upon interrupts to signal the occurrences of external events, any overhead associated with the processing of the interrupts will directly affect the system timing.

The problem is compounded if the system requires a number of interrupts to be processed at a high frequency rate. Again, benchmark testing and simulations must be conducted early in the project life cycle to determine the severity of the interrupt overhead. If the overhead is too high, then often-costly optimization approaches must be taken.

10.3.9 Prototype

many times the impact to system timing cannot be determined under simulated (mainly software-software testing) conditions. Thus, a rapid prototype of the real embedded system is required. Because the prototype is similar to the real system, the design engineer can conduct overhead and its effects on the timing of the system, the design engineer can measure and test to verify if the system can respond to events in a timely fashion. If there is a problem then corrective action can be taken.

10.3.10 Software Requirements Analysis

It is critical in the Requirements Analysis Phase for the engineering staff to have a clear understanding of what the system is required to do. Once the requirements are known, the engineer can begin to conduct feasibility studies, benchmark testing, and simulations to verify and identify those requirements that can be met and those that have risk associated with them. Because of the overhead associated with the use of Ada interrupts it is critical that in this phase engineers begin to identify how critical is the overhead in terms of meeting specific requirements. If it is determined that they cannot meet the requirements either with their current design because of Ada tool set limitations or because of the Ada language itself, corrective action must be taken early. Again, when using Ada, design engineers must be particularly careful to verify that the requirements can be met with the current Ada tool set.

10.3.11 Preliminary Design

The Preliminary Design Phase should be based on sound information developed in the Requirements Analysis Phase. If there is a lack of knowledge regarding high risk elements of the system, such as inter- rupt overhead, the preliminary design of the system can be seriously impacted. If the problem of interrupt overhead is not discovered until this phase, corrective action must be taken to reduce the risk of failure. This may mean a change in the system requirements or a modified approach to the system design. The longer a problem goes undetected the more costly it will be to resolve the problem in the later phases.

10.3.12 Detail Design

As in the Preliminary Design Phase, the Detail Design Phase must be based on sound information from the prior phases. The impact that interrupt overhead can have on the detail design of a system can result in a complete redesign of the system. This could cause major cost impacts and schedule slippages. However, if no simulation or prototypes have been built to verify the design approach, major problems, such interrupt overhead, can go unseen in the Detailed Design phase. Depending on the nature of the problem the project could be headed for failure if neither the hardware design nor the software design considers the problem.

10.3.13 Personnel Resources

With regard to Personnel Resources, the problem of interrupt overhead will not create a major problem if the risk analysis is done early in the software development cycle. However, if the problem is not identified until the Detailed Design Phase, additional personnel may be required to modify the design and stay on schedule.

If special optimization is required to the Ada tool set, specialists may be required.

10.3.14 Cost

As discussed in earlier sections of this problem (Impact of Interrupt Overhead On System Performance), major cost impact can be incurred by a project. Interrupts play a critical role in the design of embedded systems. The later in the development cycle the problem is discovered, the more costly it will be to resolve the problem.

10.3.15 Schedule

Schedule impact will be determined by how severely the interrupt overhead will affect the design of the system and how many extra resources a project must have to resolve the problem. If the problem is severe and the project lacks the required resources, schedule impact will occur. Of course, the longer the problem goes unresolved the greater the schedule slippage can be.

10.4 IMPACT OF MEMORY MANAGEMENT OVERHEAD

The run-time support provided to Ada applications programs by the Ada RSL includes a number of memory management functions. The primary functions are memory allocation and deallocation and heap storage management (garbage collection). These functions are performed by the RSL either on an as needed basis (memory allocation/deallocation during a context switch) or as required by the application (use of access types to create objects at run-time).

However, the RSL memory management features add a significant amount of run-time overhead to the performance of an Ada system. Since these functions are resident in the target machine and operate in conjunction with the application programs, they also require system resources (CPU, memory). The utilization of system resources by the RSL must be addressed when performing overall system sizing and timing analyses in the applications environment. It is important to know whether the overhead associated with the RSL memory management features is large enough to affect the ability of the system to meet specified requirements.

10.4.1 Efficiency

Use of the memory management features of the RSL can significantly increase system overhead while reducing overall system efficiency. The Ada memory management features utilize system resources - CPU time to perform the functions and storage for the memory management software within the RSL. The memory management software is resident in the target machine and operates in conjunction with the applications software. Thus, the memory management software contends with the applications programs for access to system resources (CPU and memory). Thus, when estimating or assessing system resource utilization, the resource requirements due to memory management overhead must be addressed.

10.4.2 Correctness

The RSL memory management software resides in the target environment and operates in conjunction with the applications programs. Thus, it is essential that the correctness of the memory management software for a particular application be verified as part of the overall system verification, validation, and acceptance process. This will probably require some support from the compiler vendor since the memory management software is pre-packaged and delivered as part of the RSL.

10.4.3 Verifiability

The programs which perform the Ada memory management functions reside in the target environment (within the RSL) and operate at runtime in conjunction with the applications programs. While attempting to isolate errors found during the system debugging and testing process, the performance and correct operation of the memory management software must be verified along with that of the applications programs. This can prove to be difficult, since it involves verifying and debugging software that the applications programmers are generally unfamiliar with.

10.4.4 Security

The Ada memory management functions performed by the RSL can have a significant impact on system security. The RSL memory management functions are performed at run-time in conjunction with the operation of the applications programs.

For an application which involves classified processing, it is important to control the access to classified programs and data by all software that is resident in the target environment, including the RSL memory management software. If, for example, the RSL memory management software performs garbage collection in an area of memory where classified data is stored, it is possible that the classified data could be corrupted.

Currently, it does not seem to be possible to prevent the RSL from accessing particular portions of memory (where secure information may be stored) without providing special protection methods. These methods could include the use of a software kernel or special-purpose hardware to control access to protected memory locations.

The implementation of methods such as these could require significant modifications to the RSL. Since the RSL memory management functions are pre-packaged within the RSL, these modifications would probably have to be implemented by the compiler vendor.

10.4.5 Benchmark

To develop a complete and accurate estimate of performance for an Ada system, the run-time overhead due to the memory management features must be assessed. To obtain a clear picture of expected performance for the RSL memory management functions, benchmark data should be gathered. The compiler vendor and other applications developers (if available) should be contacted to supply as much of this data as possible based on performance estimates and actual experience.

However, two main problems exist in obtaining this data. First, the data that is obtained from the compiler vendor and applications developers may not reflect the expected memory usage characteristics for the Ada system being developed. Some (extensive) interpolation will probably be required. Secondly, obtaining application-specific benchmark data for the memory management features is very difficult due to the unfamiliarity of the applications developer with the RSL code. This effort will probably require the assistance of the compiler vendor.

10.4.6 Simulation

To obtain a complete and accurate of system performance for an Ada system, an end-to-end system simulation should be performed. The model of the Ada system being developed should include the effects of the run-time memory management features. However, most Ada applications developers are not familiar enough with the operation of the RSL memory management features to develop an effective simulation. To do this will probably require support from the compiler developer.

10.4.7 Optimization

Due to the current state of the available Ada tools, the performance of Ada software systems is a major issue. The poor Ada system performance is primarily due to the inefficiency of the generated Ada code and the extensive overhead associated with the run-time functions performed by the RSL. One of the typical solutions to this problem is to perform extensive Ada system optimization.

To combat the additional overhead and potential security problems (controlled access to programs and data) due to the RSL run-time memory management features, system optimization may be required. This optimization may include modifications to the RSL memory management software to increase its execution speed, decrease its program size, and build in memory access controls. Because most applications developers are unfamiliar with the operation of this RSL, this effort would probably require support from the compiler vendor.

10.4.8 System Sizing

When performing system sizing estimates, the RSL memory management features play two important roles. First, this software resides in the target environment and operates in conjunction with the applications programs. Thus, the amount of memory space that it requires must be included in the system sizing estimates. One typical system optimization technique is to remove from the RSL those memory management features which are not required for a particular application.

Secondly, the RSL provides memory-management features which can be critical for real-time embedded systems, such as memory allocation/deallocation and heap storage management (garbage collection). For example, the memory deallocation feature could conceivably reduce the amount of system memory required to execute the applications programs by allowing the programs to deallocate memory areas as soon as they have finished processing them. But this advantage could be offset by the additional memory required to perform the memory deallocation function.

Detailed systems analysis should be performed, in conjunction with the compiler vendor, to determine the overall impact of the RSL memory management features on system sizing.

10.4.9 System Timing

When performing system timing estimates, the RSL memory management features play two important roles. First, the RSL memory management software resides in the target environment and operates at run-time in conjunction with the applications programs. Thus, the execution time required for the memory management software must be included in the system timing estimates. One typical optimization technique is to remove from the RSL those memory management functions which are not required for a particular application.

Secondly, the RSL memory management features can be critical for the development of real-time embedded applications. These features include memory allocation/deallocation and heap storage management (garbage collection). For example, performing garbage collection could conceivably improve system performance by merging non-contiguous areas of memory and providing sufficient memory to begin execution of a program earlier than would have occurred otherwise. But these advantages must be weighed against the additional system overhead associated with the memory management features.

Detailed systems analysis should be performed, in conjunction with the compiler vendor, to determine the overall impact of the RSL memory management features on system timing.

10.4.10 Vendor Interface

To ensure that the memory management functional and performance issues are addressed in the overall system design, the applications developer must establish a close relationship with the compiler vendor. This is doubly important since the sometimes extensive Ada system optimization tasks may require support from the compiler vendor and/or modifications to the compiler or RSL.

For example, the compiler can be a valuable source of memory management benchmarking data for use in performing system sizing and timing analyses. The compiler vendor can also provide support to the applications developers during system optimization and will be required to make any necessary changes to the compiler and/or RSL in conjunction with the development effort.

10.4.11 **Prototype**

To validate the system design and evaluate system performance, the application developer should build a prototype of the Ada system under development. The prototype should contain the actual system hardware and system software (compiler and RSL) and should implement critical system functions.

This prototype should be used to determine the actual system impact of the RSL memory management overhead and compare this to expected results. The prototype should also be used to evaluate the effect on memory management overhead of changes to the hardware, system software, and applications software.

10.4.12 **Compiler**

For particular Ada applications, the overhead resulting from the RSL run-time memory management features may require system optimization to be performed. This optimization may include modifications to the RSL to remove unwanted sections of the RSL memory management software, to add required functions (such as controlled memory access), or to improve its efficiency (reduced execution time and memory requirements).

If required, these changes must be designed and implemented by the compiler vendor with support from the applications developer to ensure that the changes are as requested. The changes must then be tested (via prototype) to evaluate their performance. The system documentation must be updated to reflect these changes.

10.4.13 **Software Requirements Analysis**

During the Software Requirements Analysis Phase, information concerning the performance and functionality of the RSL memory management features should be provided to the applications developers to support preliminary system sizing, timing, and functional analyses. The results of these analyses should identify potential problem areas to be addressed as part of the system design effort. If this data is not made available during the Software Requirements Analysis Phase, the results of the system analyses will not be accurate. This could result in severe problems during the system design and implementation

phases in areas such as design rework, requirements for additional hardware and software, extensive system optimization, and system security.

10.4.14 Preliminary Design

During the preliminary design, the result of the software requirements analysis efforts are used to develop a preliminary system design. The performance and functional capabilities of the RSL memory management software must be made available to the system designers to support their efforts. This information is evaluated to determine its impact on system hardware selection, system sizing and timing estimates, and selection of an Ada compiler and support software tools. The results of the preliminary design efforts may also identify potential areas where the RSL memory management features must be modified to meet system requirements. If these areas are not identified as early as possible in the system development cycle, they could critically impact the entire project, since they must be implemented by the compiler vendor in parallel with the development effort.

10.4.15 Detail Design

During the detailed design effort, detailed system sizing and timing estimates should be available which include the overhead due to the memory management features. Any required design changes to the compiler and RSL memory management software should be incorporated into the overall system design. A prototype of the Ada system should be built to verify the functional capabilities and the accuracy of the sizing and timing estimates for the RSL memory management software.

If these activities are not performed, the implementation and testing efforts could be impacted due to performance problems related to the memory management features. This could result in extensive design rework in order to meet system performance requirements.

10.5 IMPACT OF RUN-TIME SUPPORT LIBRARY OVERHEAD ON SYSTEM PERFORMANCE

The Runtime Support Library (RSL) is the total package of software required at run-time to support the execution of the object code generated by the compiler for the application program. Functions such as dynamic memory management, system activation/allocation, interrupt processing, input/output operations, co-processor support, and tasking are all performed by the RSL. The basic problem with the RSLs of today is that they are generally too large and too slow for many embedded system applications. In terms of sizing, the problem is more noticeable when the application program is fairly small. In this case, it's very possible that the RSL can be larger than the application program. As the application programs grow larger, the proportion of memory taken by the RSL become less, thus the impact is not as great. For timing impacts, the amount of overhead experienced will depend upon what features of the language are used. Generally speaking, the more you use the unique features (tasking, dynamic memory, delays, etc.) of Ada, the more over-head that is incurred. This can be attributed in part to the immaturity of existing RSLs. Because some of the Ada features are new to the implementors of the language, the techniques of implementing them efficiently are still emerging.

As an example of what some contractors are experiencing in RSL overhead, on Sonicraft's MEECN (Minimum Essential Emergency Communications Network) project it was discovered that the RSL alone would require over ninety kilobytes (KB). Of course, optimization efforts had to begin immediately to reduce size since the system was limited in its memory and power usage.

10.5.1 Efficiency

The overhead associated with the Run-Time Support Library will impact the efficiency of embedded systems in terms of sizing and timing. Because of the inefficiencies of the RSL, embedded systems today will utilize more memory (RAM, ROM) and execute programs slower.

10.5.2 Benchmark

Good benchmark testing of the Run-time Support Library (RSL) is a must to insure the successful design and development of an embedded system using Ada. All aspects of RSL overhead problems must be identified and done so as early in the software development cycle as possible. For without this critical information designers will design the system based on faulty information or the lack of information. This can lead to major cost and schedule impacts, if not the complete failure of the

system to performance. If the contractor cannot obtain the required benchmark test results from the developer of the Ada tool set, they must conduct benchmark testing themselves.

10.5.3 Simulation

To verify that the RSL overhead problems will not cause a problem in one's design approach, one must conduct simulations of the system. This will help identify the RSL overhead problems that are a risk to the project. Problems such a sizing and timing are critical to the development of the system. If the sizing and timing requirements of the RSL are not discovered until the Detailed Design Phase major cost and schedule impact can be incurred by both software and hardware.

10.5.4 Optimization

If through benchmark testing and simulation it is discovered that optimization is required, one may find this to be very expensive. This is because the *Run-Time Support Library* is larger and more complex than in other high languages. Usually the contractors cannot modify the RSL themselves, so they have to contract the vendor to perform the optimizations.

When Sonicraft discovered that the size of the RSL was over ninety kilobytes, we began to investigate optimization solutions. It was determined that one of the major size drivers was the fact that major portions of the RSL was written in Ada. Since the Ada compiler was very inefficient it produced a very large RSL. The solution was to rewrite much of the RSL in assembly language.

10.5.5 System Sizing

The problem of having *RSL overhead* will directly affect the system sizing. Because the RSL is inefficient it will require more code to perform a particular function. This means more memory is required for the system. To avoid a situation where the application program will not fit into the available system memory, one must conduct simulations and possibly build a rapid prototype. By doing so, problems will be identified early in the development cycle and corrective action can be taken.

10.5.6 System Timing

To determine how the RSL overhead problems will affect the system timing requirements, one will have to built a rapid prototype of the real embedded system. This will allow the design engineers to examine the timing of the system and determine what problems *may* exist. As soon as problems with regard to timing are known, the chances the project will have for successful completion within cost and schedule can be known.

A key problem that affects the system timing is the RSL's ability to process interrupts efficiently. Please refer to problem number three for a more detailed discussion of this problem.

10.5.7 Prototype

As referred to in the System Timing section for this problem a rapid prototype may be required to identify what impacts the RSL overhead problems will have on the requirements of a particular system. The earlier the problems are identified the sooner corrective action can be taken to resolve the problem.

10.5.8 Compiler

There is a direct connection between the overhead problems of the Run-Time Support Library and the Ada Compiler. The connection is that much of the RSL is written in Ada and, as discussed in problem number seven, the Ada compilers of today produce inefficient object code. So when the run-time code is compiled by an inefficient compiler, the result is a large, inefficient RSL. As an optimization solution to the problem of having a large RSL, Sonicraft contracted the vendor to rewrite portions of the RSL in Intel assembly language. This way less of the RSL code was produced by the Ada compiler and some degree of size reduction was obtained.

10.5.9 Software Requirements Analysis

In the Software Requirements Analysis Phase the engineering staff must have a clear understanding of what the system is required to do. Once the requirements are known, the engineer can begin conducting feasibility studies, benchmark testing, and simulations to verify and identify those requirements that can be met and those that have risk associated with them. The overhead impacts associated with the Run-time Support Library make it critical that in this phase engineers begin to identify how critical is the overhead in terms of meeting the system requirements in software vs. hardware. If it is determined that they cannot meet the requirements because of the current design, the RSL, or the Ada language itself, corrective action must be taken early. This task may demand that training to understand RSL operation be obtained.

10.5.10 Preliminary Design

The Preliminary Design Phase should be based on sound information developed in the Software Requirements Analysis Phase. A lack of knowledge regarding high risk elements of Ada, such as run-time support impacts, can jeopardize the preliminary design of the system. If the problems of run-time

overhead are not discovered until this phase, it will be more costly to take needed corrective action. This may mean a change in the system requirements or a redesign of the run-time code. The longer a problem goes undetected the more costly it will be to resolve the problem in the later phases.

10.5.11 Detail Design

The Detail Design Phase must be based on sound information from the prior two phases. The impact that run-time overhead can have on the detail design of a system can result in the complete failure of the system to perform to specifications. This can cause major cost impacts and schedule slippages. Again, if no simulations or prototypes have been built to verify the design approach, major problems, such as run-time overhead problems, can go unseen in the Detailed Design Phase. Depending on the nature of the problems the project could be headed for failure since it is the RSL that is controlling most of the execution of the application program at run-time.

10.6 IMPACT OF TASKING OVERHEAD ON SYSTEM PERFORMANCE

One of the key features of the Ada language is tasking. Ada tasks are "entities whose executions proceed in parallel [REF. 1]."This feature gives Ada a great advantage or other high-level languages, but not without a price. The cost is in terms of overhead. Tasking overhead affects the efficiency of the system in both sizing and timing.

Whenever a designer decides to utilize tasking in an Ada program, he will automatically incur an additional cost in terms of additional run-time support code, which can be as high as thirty kilobytes. This code is required to perform the various features (entry calls, accepts, selects,..etc.) of Ada tasking at run-time. Another sizing problem has to do with the stack requirements of tasks. The designer must allocate enough memory for his application to make available the additional stack memory for task control information. Also, any stack memory required for any run-time procedures called to execute a particular feature must be added to the total size of the task stack allocation. The stack allocation requirements are required for each task declared in the designer's application program. Thus, the problem is compounded.

With the use of tasking, today's applications will experience timing overhead impacts due to tasking features like task allocation, task activation/termination, task switching, synchronization and task rendezvous. To determine what kind of overhead would be incurred by using tasking, a study was performed by Hughes Aircraft Company. The study conducted a series of tests using the DEC Ada Compiler (1.2) on a VAX 8600 (VMS 4.2). The following results show the magnitude of task overhead compared to the processing done within the task itself.

Description	Task Overhead (usec)	Normal Proc. (usec)
1. Task activation and termination	1960	178
2. Task created via an allocator	150	14
3. Producer-Consumer (2 task switches)	503	46
4. Producer-Buffer-Consumer	1220	111
5. Producer-Buffer-Transporter-Consumer	1694	154
6. Producer-Transpt-Buffer-Transpt-Consumer	2248	204
7. Relay	906	82
8. Conditional Entry		
- no rendezvous	170	15
- rendezvous	29	3
9. Timed Entry		
- no rendezvous	254	23
- with rendezvous	33	3
10. Selective Wait with Terminate	127	12
11. Exception during a rendezvous	962	87

[Ref] T. M. Burger and K. W. Nielsen, "An Assessment of the Overhead Associated with Tasking Facilities and Task Paradigms in ADA", Hughes Aircraft Company.

10.6.1 Efficiency

Because of the tasking overhead in Ada today, many embedded systems will try to avoid the inefficiencies by minimizing or even eliminating the use of tasking. By doing so, the code at the Ada source level may be inefficient code, since the tasking features of Ada would be better suited for these particular functions. However, the non-tasking code may produce smaller amounts of code to run on the target machine because this code need handle only the special case at hand.

10.6.2 Benchmark

Benchmark testing of tasks has proven to be difficult, partially because of compiler vendors and their implementations of tasking. The compiler implementors have been given a great deal of implementation latitude, and as a result, it is difficult to develop a set of benchmarks that completely characterize this area. However, difficult it is for one to test the characteristics of tasking, the information is urgently needed if embedded systems are be developed without unforeseen impacts.

10.6.3 Simulation

Simulations of embedded system designs using Ada tasking is required for a number of reasons. First, the concept and use of Ada tasking in embedded system design is new to many of today's Software engineers. As a result, the impact of using tasks can go without notice and result in major cost impacts.

Second, the characteristics of the compiler with regards to tasking may be vague or unknown, thus the simulation will help identify problems associated with the use of tasking.

Third, the simulation provides a time where the design can be optimized to avoid the overhead impacts of tasking.

On Sonicraft's MEECN project simulations were conducted to determine how to control the task elaboration and activation process. This was necessary to insure the proper start-up of the system. In the process, it was discovered that the task elaboration/activation overhead time required to complete start-up was longer then the time period for the system power-up-fail indicator, as a result system design modifications were required.

10.6.4 Optimization

The inability of todays RSLs to handle tasking efficiently will impact many embedded system projects to the point of requiring special optimization. Such was the case on Sonicraft's MEECN project, an Intel 8086 based system, where it was discovered that task switching times were on the order of milliseconds. Special optimization of Ada compilers will often require the assistance of the vendor. Of course, this increases the cost to develop the system using Ada.

10.6.5 System Sizing

Tasking overhead will directly affect the sizing of the system. The use of tasking in the system will require additional run-time support code, thus more ROM memory is required. The inefficient use of stack space will also require additional RAM. An example of this is the "Null Task" used in the 8086 ALS. The purpose of this task is to execute when no other task is available to execute. This task contains virtually no code and therefore it should need very little RAM. However, since it's a predefined task it is assigned the default stack size of four kilobytes plus two kilobytes additional space for the handling of errors. Thus, you have six kilobytes allocated for a task that does virtually nothing.

10.6.6 System Timing

As in system sizing, the overhead of tasking will directly affect the system timing. As the number of tasks increase in the design so will the overhead. This is one of the major reasons why embedded system designers try to avoid using tasking: the overhead costs both in terms of sizing and timing are too high.

10.6.7 Prototype

As in problem number five, the development of a rapid prototype to prove or disprove the design; is an excellent way of determining how tasking overhead will impact the performance of the system. Although prototypes can be expensive, the cost savings obtainable due to the risks associated with tasking makes it a viable option.

10.6.8 Software Requirements Analysis

With the commitment of DOD to the use of the Ada language,contractors must perform a comprehensive analysis of their requirements. Many of today's requirement specifications are written with the mind set of past performance in embedded systems. However, with the Ada mandate firmly supported, the immaturity of Ada compilers versus the performance expectations of these spec writers may be a problem for now. The impact of tasking overhead requires the system to have more memory, and execution speeds are currently slow in areas such as task switching. These impacts may require that the system requirements be expanded to better support the use of Ada. Other options may be to continue development under the restrictive requirements and contract to have the tasking overhead problem optimized. Clearly, it is critical that the system requirements be closely reviewed with respect to the use of Ada.

10.6.9 Preliminary Design

The impact that tasking overhead can have on the development of a preliminary design is severe. Because this is the case engineers cannot assume that Ada tasking will execute with the speeds they are accustomed to. Early in the Preliminay Design Phase sizing and timing analyses must be conducted to reduce risk. These analyses must be based on sound benchmark results on the compiler and simulation results on the system's critical functions. By doing so, the impact of tasking overhead can be considered in the preliminary design of the system.

10.6.10 Detail Design

At this point in the development cycle the impact of tasking overhead can cause a complete redesign of the system. Tasking is one of the main program units which directly affects system timing. Considering that interrupts are handled by tasks, the degree to which a real-time embedded system is capable of responding to real-time events; will be determined in part by the impact that tasking overhead has on the system. Again, analysis should be conducted prior to this phase to avoid major cost and schedule impacts.

10.7 INEFFICIENCY OF OBJECT CODE GENERATED BY ADA COMPILERS

As with most first generation compilers for new languages, the Ada compilers today are somewhat inefficient because of the complexity of the Ada language. Unfortunately, the lack of efficient compilers directly impacts the development of embedded systems today. Embedded systems typically have very restrictive requirements on sizing, timing and power consumption. Therefore, any inefficiencies in the object code generation will impact the cost and performance of the typical weapon system. Because the compilers are producing more code than required to implement a particular function the compilation time is longer. As a result, it takes developers somewhat longer to complete the coding process. This means schedule and cost impacts.

10.7.1 Efficiency

The inefficiency of Ada compilers directly affects the efficient execution of the run-time program. Embedded system development is impacted both in terms of memory usage and execution speed. One of the chief contributors to this problem is the validation process. While "validation guarantees legitimate Ada code, it does not guarantee performance" [REF 1]. As a result there are many validated compilers today that are production quality, with production quality meaning: a compiler capable of producing machine code that does not greatly exceed that obtained from assembly-language programming, and the execution speed of this code must be comparable to that of assembly-language machine code.

[Ref] W. Myers, "Ada: First Users - Pleased; Prospective Users - Still Hesitant", Computer, 1984.

10.7.2 Benchmark

As a consequence of the lack of high quality ACVC tests to provide embedded system designers with any test results on the performance characteristics of Ada compilers, it has been necessary for the embedded designer to either develop or contract the development of benchmark testing of Ada compilers.

10.7.3 Simulation

Due to the impact of the inefficient object code generated by todays Ada compilers, embedded system designers must develop and conduct comprehensive simulations of their critical functions to verify they can meet the requirements.

10.7.4 Optimization

The impact that the inefficient compiler has on an embedded project is the requirement to optimize. Either the embedded designer spends time optimizing his design as a consequence of the compiler or the compiler itself is optimized. Both can be very expensive.

10.7.5 System Sizing

System sizing is directly impacted by the inefficient generation of object code. This will impact the amount of memory required in the system, power consumption and system reliability. Whenever possible the designers may take shortcuts, such as the inclusion of assembly language, to obtain a reduction in memory, which can lead to less maintainable code.

10.7.6 System Timing

Inefficient object code mainly affects system timing due to added overhead time caused by the execution of the unneeded code. In some cases the added overhead can cause the system to perform poorly in response to real-time events.

10.7.7 Compiler

The problem of the inefficient Ada compiler is probably the most discussed problem in the Ada community. The reason for this is that the compiler is one of the first (and one of the most used) tools needed to make the software a reality. If the compiler does not work properly or works inefficiently, the project is immediately impacted. Such is the case with many of the validated compilers today. Some of the common problems are; slow compilation speed, poor diagnostics, failure of valid Ada constructs and inefficient object code generation.

10.8 NEED FOR EXTENSIVE ADA OPTIMIZATION

Due to the inefficiency of compiler-generated Ada applications code and the excessive overhead associated with the RSL run-time support, extensive optimization is generally required to improve the performance of Ada systems.

The types of system optimization that are performed include:

* Customizing the RSL for a particular application
* Reducing RSL overhead (tasking, memory management, interrupts)
* Adding special-purpose hardware and software
* Rewriting applications programs (algorithms)
* Modifying compiler implementation details
* Using non-Ada software
* Providing absolute addressing capability
* Providing for storage of constants in ROM

However, this extensive optimization can adversely affect system performance and project productivity. The addition of special-purpose hardware and software along with the use of project-specific programs can reduce system portability, reliability, maintainability, reusability, and verifiability, and can increase system complexity. Also, the extensive rework that is performed as part of the optimization efforts can decrease overall project productivity.

10.8.1 Reliability

As a result of the extensive optimization that is sometimes required during the development of Ada systems, the overall system reliability can be reduced. This is due to the fact that system changes, particularly when made to a complex system, can impact other areas of the system in subtle ways that are only discernible under certain conditions. System problems that are related to these changes may only occur in times of system stress or when a very specific series of events occurs. If these situations are not identified and corrected, the overall system reliability is reduced. Comprehensive regression testing must be performed as part of the system optimization process to minimize the adverse impact on system reliability.

10.8.2 Maintenance

As a result of the extensive system optimization that is performed during the development of real-time embedded Ada systems, overall system maintainability can be reduced. This is due to the increase in overall system complexity that generally accompanies the optimization process. The increase in system complexity makes the operation of the system more difficult to understand and thus reduces system maintainability.

10.8.3 Verifiability

The extensive system optimization that is sometimes performed in the development of Ada systems can adversely impact the verifiability of the system. This is due to the increase in system complexity that occurs as a result of the optimization efforts. The increase in system complexity makes it more difficulty to test the system and to verify that the system is meeting its requirements.

10.8.4 Portability

The extensive system optimization that is performed during the development of Ada systems can reduce system portability. A number of the changes that are implemented during the system optimization process provide enhanced performance or functionality for a specific application. This is usually done by taking advantage of implementation dependencies in the target machine or system software or by adding special-purpose hardware and software to the system. Overall, these changes make it more difficult to transport the developed Ada software to another target machine or to use a different set of Ada tools (off-the-shelf).

10.8.5 Complexity

The extensive system optimization that is performed during the development of Ada systems can increase the overall complexity of the system. This is due in part to the patching that must be done to the original design to implement changes, particularly if the system is not designed in a modular fashion to accommodate such changes. The requirement for special-purpose hardware and software to perform the optimization also contributes to the increase in system complexity.

10.8.6 Reusability

The extensive system optimization that is performed during the development of Ada systems can reduce the reusability of the system and applications software. This is because a number of the changes that are made to support the optimization efforts are project-specific. These changes are implemented by modifying the applications and system software to take advantage of design and implementation details that are specific to the application being developed and to any special-purpose hardware and software that are used in the application. The overall effect is to reduce the reusability of the applications and system software.

10.8.7 Development Environment

The extensive optimization performed during the development of Ada systems can affect the applications development environment. This occurs when the proposed optimization approach requires the use of special-purpose hardware or additional system/support software. Drastic changes to the development environment can have a major impact on the overall system performance and characteristics and can also adversely impact the system development schedule.

10.8.8 Non-Ada Software

To implement the extensive optimization that is performed during the development of Ada systems, the use of non-Ada software may be required. This non-Ada software is generally used to improve system performance (sizing and timing) or to implement machine-dependent functions (low-level hardware interfaces). However, the use of non-Ada software, particularly assembly language, tends to reduce the overall maintainability of the system, due to the fact that most non-Ada languages cannot provide the readability, understandability, and power of Ada.

10.8.9 Vendor Interface

In the current Ada development environment, a large amount of the system optimization that is performed involves modifications to the Ada compiler and RSL. Thus, it is critical that a working interface be established between the applications developer and the compiler vendor. The compiler vendor should provide information to the applications developer concerning the operation and performance of the Ada tools that are supplied (consulting support) and should also implement any required changes to these tools.

10.8.10 Compiler

In the current Ada development environment, a large number of the changes that are implemented during optimization are made to the compiler and RSL. The complexity of the compiler makes it essential that extensive regression testing is performed while implementing these changes. It is easily conceivable that implementing a change to one portion of the compiler could impact other sections of the compiler, thus possibly reducing the overall correctness and reliability of the compiler.

10.8.11 Preliminary Design

During the Preliminary Design Phase, the initial optimization methods that are necessary to meet system requirements should be identified. This optimization can result in changes to the design of the applications program and also to the design of the Ada software tools (compiler, etc.). Those optimization requirements that are not identified in this phase must then be identified in later phases, where a larger amount of effort is required to rework the design.

10.8.12 Detail Design

During the Detailed Design Phase, those optimization requirements that were not identified during the Preliminary Design Phase must be addressed. The optimization efforts may involve changes to the applications programs and the Ada system software tools (compiler, etc.). Any optimization requirements that are not addressed in this phase must be addressed in later phases, where the effort required to rework the design is much more extensive. Also, any requirements for special-purpose hardware and software should be identified in this phase and incorporated into the overall system design.

10.8.13 CSC Test

During the CSC Test phase, the additional system complexity that results from the extensive Ada system optimization increases the effort required to test and verify CSC performance and operation. Also, any additional system optimization tasks that are identified in this phase will require more effort to implement because they were found so late in the system development cycle.

10.8.14 CSCI Test

During the CSCI Test phase, the additional system complexity which results from the extensive Ada system optimization increases the effort required to test and verify CSCI operation and performance. Also, any additional system optimization tasks that are identified in this phase will require more effort to implement because they were found so late in the system development cycle.

10.8.15 Personnel Resources

To perform the extensive optimization that is required to develop Ada systems, a larger number of senior engineers are required. These engineers must have experience in the design and implementation of Ada systems and must have some knowledge of the operation and performance of Ada system tools. These personnel requirements should be supplemented by obtaining consulting support from the compiler vendor.

10.8.16 Facilities

The extensive optimization that is required to develop Ada systems can result in a requirement for additional development and test equipment and facilities. The additional equipment could include special-purpose hardware, system software, and support tools. The additional facilities could include sophisticated emulation and debugging systems to address the increased complexity of the applications programs.

10.8.17 Cost

The extensive optimization performed during the development of Ada systems can add significantly to the system development cost. The development costs can increase significantly if the proposed system changes result in extensive rework to the system design and implementation. The development cost can be further increased if the optimization requirements are not identified until late in the system development cycle, since this increases the amount of rework that is required.

10.8.18 Schedule

The extensive optimization that is performed during the development of Ada systems can have a significant impact on the system development schedule. The development schedule can be lengthened due to the extensive amount of effort required to rework the system design and implementation. The development schedule can be further impacted if the optimization requirements are not identified until late in the system development cycle.

10.8.19 Estimation

To ensure that the proposed system development cost and schedule are as complete and accurate as possible, the estimated effort required to perform Ada system optimization should be included in the overall project estimates. This data can be estimated using historical data, if it is available, or by using a reasonable estimate for the particular type of Ada application being developed, based on Ada software cost estimation models. However, it should be noted that, currently, very little historical data exists for Ada projects and very few Ada cost estimation models exist which have been proven to be accurate.

10.9 INADEQUATE DEBUGGING CAPABILITIES PROVIDED BY CURRENT DEBUGGERS

Poor debugging tools do not give the engineer adequate control and visibility of the program at run-time. This will directly affect the verifiability of the program and the system.

10.9.1 Debugger

The debugger plays a major role in the finalization of a software development effort. An inadequate debugger can cause long delays in the finalization process. Thus, schedules are slipped and cost impacts are incurred. Because the designer will now have to debug a system that contains code not developed by himself, the problem of an inadequate debugger is compounded. Such is the case with Ada debuggers that do not provide the necessary control and visibility to efficiently debug an Ada program. Hence, the designer may be forced to contract the vendor of the run-time support library to help debug his system.

10.10 ADA EXCEPTION HANDLING

The whole idea of handling unexpected errors at run-time is a very good concept. But having developed and debugged some Ada programs, one begins to sense there is a problem with the handling of exceptions. The problem has to do with the manner in which an exception is reported to the engineer and the lack of information that is conveyed. This is particularly true when the exception that is raised is a non-user-defined exception. Further, if the application program does not require the inclusion of TEXT_IO, and many will not because of its large size, or no capability to display text messages exists in the run-time program, the problem can be compounded. This is because it is now possible for an exception to be raised and the engineer not know of its existence until it manifests itself as a failed function much later. Now the engineer must search for the source of the exception and then determine why the exception was raised. Depending on the debugging tools available to the engineer and how far the manifestation of the problem is from the real problem, the determination of the problem can be long, frustrating and costly.

10.10.1 Verifiability

The inability of today's run-time support code to generate adequate trace-back information when exceptions occur will directly affect the verifiability of Ada programs. Without verification of proper execution, bugs can remain in programs and cause future problems in the application system.

10.11 IMPACT OF EXTENSIVE USE OF GENERICS

The use of Ada generics is regarded as a major step towards the development of reusable Ada programs. However, in the current Ada environment, extensive use of Ada generics can adversely impact the performance of Ada systems and the productivity of Ada development efforts. The extensive use of generics can result in a significant increase in system memory requirements. This is due to the general inefficiency of code that must be shared by a variety of programs, the additional code that must be required to implement conditional processing within the shared code, and the additional processing required to implement the use of the actual parameters within the generic instantiation.

There are also impacts on productivity. Each time a generic program is changed, all programs which contain an instantiation of the generic must be recompiled. Also, the generics instantiations are essentially compiled as in-line code, which increases overall compile time.

10.11.1 Efficiency

The extensive use of generics can have an adverse impact on system efficiency. One reason for this adverse impact is that code designed to be used (shared) by number of different programs is generally written less efficiently than code that is only to be used for one specific purpose. The algorithms for shared code are usually written in a more general (flexible) fashion and the shared code usually involves some amount of conditional processing, which also reduces efficiency.

Because of the way in which Ada implements generics, a substantial amount of code which supports the actual parameters for each generic instantiation is included in the memory requirements for generics. Also, in some implementations of the Ada compiler, generics are, in effect, compiled as in-line code. This can greatly increase the system memory requirements.

10.11.2 Benchmark

To obtain an estimate of the impact on system memory requirements on the extensive use of generics, it is important to benchmark the performance of the compiler in generating the generic instantations. This information is useful in assessing the overall system resource requirements.

10.11.3 Optimization

The extensive use of generics could cause a significant increase in system memory requirements. Minimizing the impact of these increased memory requirements could require system optimization. This optimization could include modifying the processing performed by the generic to require less memory or modifying the compiler to reduce the memory requirements for generic instantiations.

10.11.4 System Sizing

The extensive use of generics could cause a significant increase in the overall system memory requirements. This impact must be considered when performing system sizing estimates.

10.11.5 Prototype

The extensive use of generics can cause significant increases in the system memory requirements. To obtain an assessment of the actual system impact and to evaluate the potential memory savings due to the proposed system optimization techniques, a prototype of the system should be built. The information obtained from the prototyping effort provides valuable input to the system design.

10.11.6 Cost

The increased system memory requirements that result from extensive use of generics and the effort required to perform system optimization can increase the overall system development cost. Overall project productivity is decreased because each time a generic is modified and recompiled, all programs which instantiate that generic must be recompiled. Also, the generics are often compiled as if they were in-line code, which increases overall system compilation time.

10.12 INABILITY TO PERFORM INDEPENDENTLY OF THE RSL

One of the common requirements for the embedded system is the requirement to perform Built In Test (BIT). BIT is the ability of the embedded system to perform a self-test without external equipment and indicate if the system is good or bad. BIT is usually performed by some combination of hardware and software. One of the key functions in performing a self-test is the setting of the system to a known state. A good example of this would be the initialization of RAM (random access memory). When an embedded system is first powered up, the state of RAM is unknown. Therefore, one of the functions of BIT is to set RAM to a known state and then verify it. The problem occurs when the application is implemented in the Ada language. The run-time support code is designed to take control of the system at power-up and perform system elaboration. When elaboration is completed all task stacks and variables have been allocated. But the state of memory (RAM) is still indeterminate. If BIT were to run after elaboration it would destroy the state set up by the RSL. Therefore, BIT must execute before the RSL. Today, one must modify the run-time support code to perform this type of function, and to do so can be very costly and time consuming because you are modifying someone else's code.

10.12.1 Reliability

The inability to perform BIT befre the RSL configures processor memory means that much of RAM is untestable. This would have a severe impact on system reliability because a RAM failure could go undetected for some time, causing erroneous operation of the system. Since this unacceptable, some other scheme is necessary as a work-around, such as RAM bank seitching or an assembly language routine that runs before the RSL takes over. Bank switching is usually not an usually not an option due to the system penalties when additional computer resources are used.

10.12.2 Vendor Interface

The use of an assembly language routine to perform BIT before the RSL configures processor memory can require extensive vendor interface. This is because many RSLs are currently unable to surrender control grcefully, as BIT requires. Where the ability to seize control is designed into the RSL this aspect of the problem goes away.

10.12.3 IMPORTER

The use of an assembly language routine to perform BIT before the RSL configures processor memory requires some provision for importing code written in a language other than Ada. This capability is usually present in Ada compillers now on the market so this is rarely a problem.

10.12.4 Preliminary Design

The use of an assembly language routine to perform BIT before the RSL configures processor memory impacts the Preliminary Design phase of the development cycle. During this phase the compiler must be selected, and the requirements for an importer [refer to Importer discussion for this problem] and for vendor interfce to assuure a way to seize control from the RSL [refer to Vendor Interface discussion for this problem] should be a big consideration in its selection. Failure to address these needs will impact subsequent design phases very severely.

10.13 LACK OF A DISTRIBUTED RUN-TIME SUPPORT LIBRARY (RSL).

The term "concurrent processing" is often mentioned in the same breath as Ada. This is because the Ada is designed with concurrent processing as a goal. This goal is met by creation of tasks. However, many of today's implementations of the Ada Language Reference Manual (LRM) are not capable of true parallel (concurrent) processing. This can be attributed to the lack of RSLs that are designed to be distributed across multiple processors. Another reason may be the LRM itself, since it does not address the requirements for distributed Ada processing. Many of today's embedded system applications have requirements that warrant the design of systems capable of true parallel processing. Where parallel processing is required, the technique that is often implemented is distributed processing. Distributed processing occurs when computer processes are distributed across multiple computer processing units (CPU) to achieve true parallel processing. Today, an embedded system with multiple processors, implemented in the Ada language, must develop Ada programs for each CPU in the system. Thus, the system incurs the cost of additional memory, timing, and hardware to accommodate an RSL for each processor.

10.13.1 Efficiency

Lack of a distributed RSL hurts memory efficiency by forcing a separate copy of the RSL in each processor which is using Ada instructions. Strong coupling to a central memory bank containing the RSL conflict arbitration schemes not available in current Ada compilers.

10.13.2 Integrity

Lack of a distributed RSL hurts software integrity because processors, being loosely coupled, cannot be built up in a hierarchically secure manner. The most secure systems ("trusted" systems) have an almost impenetrable single core, which is impossible in a distributed Ada system today without extensive modification to the RSL.

10.13.3 System Sizing

Lack of a distributed RSL increases system sizing because multiple copies of the RSL are required. Even in a modest system of three or four processors this can add on the order of 100 kilobytes to the memory requirement (depending on the compiler selected). For complex systems, such as modern avionics with over a hundred processors, this penlty can be severe enough to redesign the RSL to eliminate the multiple copies.

10.14 DIFFICULTY IN PERFORMING SECURE PROCESSING FOR ADA SYSTEMS

Due to the current unavailability of a secure Ada operating system and the excessive overhead associated with the use of a secure Ada kernel to restrict system memory accesses, it is currently difficult to build a secure processing application in Ada.

The Ada language allows the applications programmer to perform run-time, and system level operations which increase the difficulty involved in protecting classified programs and data within the system. These operations include the creation, access, and destruction of objects at run-time, the use of address specifications to access particular memory locations; and the writing and execution of in-line assembly language programs.

Also, the RSL code is not only resident in the target environment, but runs in conjunction with the applications programs. Thus, to fully verify the security of an Ada system, the RSL code must be evaluated and certified as part of the system certification effort. This is difficult because most Ada applications programmers have little knowledge concerning the operation of the RSL.

10.14.1 Integrity

One of the major problems in developing Ada systems to perform secure processing is to maintain system integrity (programs and data). Ada is a very powerful language which allows an applications programmer to perform low-level activities such as creating, accessing, and destroying objects at run-time; using address specifications to access specific memory locations; and writing and executing in-line assembly language programs. The Ada run-time environment also contains the RSL which operates in conjunction with the applications programs and accesses the system data in memory.

Capabilities such as these make it much more difficult to ensure the integrity of system programs and data since they can be performed at run-time by the applications programmer or by the system software (RSL). Currently, there are no secure operating systems available for use in Ada systems, and the overhead involved with implementing a secure kernel is prohibitive for most Ada applications. Thus, these security issues are generally addressed at the program level and by the use of special-purpose hardware which provides the required protection.

10.14.2 Security

For Ada systems in which secure processing must be performed, the difficulty required to construct a secure Ada environment can be a major source of problems. Currently, there are no secure operating systems for use in the development of secure Ada applications, and the overhead associated with the use of a secure kernel is generally considered too prohibitive for current Ada applications.

The security level of the Ada applications programs can be monitored reasonably well by reviewing the applications programs at both the source and object code levels. However, this task is made more difficult by the availability of certain Ada features that can be used by the Ada applications developer at run-time. These features include creating, accessing, and deleting objects; using address specifications to access specific memory locations; and writing and executing in-line assembly code.

Another security problem is created because the RSL programs are resident in memory and operate in conjunction with the applications programs. To properly ensure system security, the RSL programs must also be verified and certified to meet system security requirements.

10.14.3 Complexity

Currently, there is a lack of secure operating systems for Ada applications and the overhead associated with building a secure kernel is considered to be prohibitive for most Ada applications. Thus, Ada applications developers must generally use special-purpose hardware and software to implement the security mechanisms required to protect the system programs and data. However, this special-purpose hardware and software can increase system complexity through the addition of sophisticated and complex hardware and software interfaces.

10.14.4 Risk

The security risks associated with developing a secure Ada system in- involve two major areas.

The first area is providing protection to offset the powerful run-time memory access and control capabilities provided to the applications programmer by the RSL. Standards and procedures must be established to monitor the usage of these functions. If this is not done properly, it is possible that the system might not receive certification.

The second area is to verify and certify the RSL software which resides in the target environment and operates in conjunction with the applications software. This is risky because it must

be done in the context of the application environment (hardware, software, optimization) to achieve system certification, and any problems that are found could require a significant effort to correct (special-purpose hardware and software, system redesign).

10.14.5 Development Environment

Currently, due to the lack of a secure Ada operating system and the overhead associated with developing a secure kernel, security requirements are generally implemented through the use of special-purpose hardware and software. The addition of this special-purpose hardware and software and the overhead associated with performing the security can have a significant impact on the development environment. It may require the redesign of system interfaces and optimization to provide system performance enhancements. Also, the security requirements and the proposed method of implementation are a major factor in the choice of system hardware, software, and support tools.

10.14.6 Vendor Interface

Due to the lack of a secure Ada operating system and the overhead associated with developing a secure kernel, a number of the system modifications that must be made to meet security requirements involve changes to the system and support software (compiler, RSL, debugger, etc..). To implement these changes, the applications developers will require extensive support from the compiler developers. Thus, a fairly flexible interface must be established between the compiler vendors and the Ada applications developers.

10.14.7 Prototype

To ensure that the security measures required to support the development and operation of an Ada system have been properly designed and implemented, a prototype should be built and used to verify their correctness. The prototype will also provide information concerning the impact on system performance due to the overhead associated with performing the required system security functions.

10.14.8 Compiler

Due to the lack of a secure Ada operating system and the excessive overhead associated with a secure kernel, a number of the system changes that are implemented to meet security requirements involve modifications to the compiler and RSL. These modifications might include adding software to the RSL to provide an interface to any special-purpose security hardware and software in the system.

It is important to assess the impact of these changes on other

parts of the compiler and RSL and on the applications programs. It should be noted that if modifications such as these are made to the compiler and RSL after they have been certified, the software must then be recertified before it can become part of the system.

10.14.9 Software Requirements Analysis

During the Software Requirements Analysis Phase, the system security needs must be reviewed to determine which ones can be achieved based on the security capabilities of the Ada development environment. Those security requirements which cannot be met (due to the lack of a secure Ada operating system or to the prohibitive overhead associated with a secure kernel) must be identified for resolution later on in the system development cycle. Failure to identify security issues early on could result in extensive system design rework in later development phases.

10.14.10 Preliminary Design

In the Preliminary Design Phase, those security issues that were identified in the Software Requirements Analysis Phase must be addressed. Solutions must be designed and incorporated into the overall system design. If the solutions require special-purpose hardware/software or changes to the system software (compiler, RSL, etc.), then these issues should be initially addressed during this phase. If these issues are ignored until later development phases, the amount of effort required to rework the system design increases significantly.

10.14.11 Personnel Resources

To develop a secure Ada application, the required personnel qualifications are much more intensive than for a non-secure application. The project personnel must be experienced in the design and development of secure systems, including the use of special-purpose hardware and software to implement security requirements. They must also be familiar with the security limitations placed on the system based on the choice of an Ada compiler and RSL for a particular application. This knowledge should be supplemented by technical support from the compiler vendor.

Personnel with this type of background are hard to find. However, if these people cannot be found, the project runs the risk of not being completely responsive to the system security requirements, thus making it difficult to obtain system certification.

10.14.12 Cost

The workarounds that are required due to the lack of a secure Ada operating system and the excessive overhead associated with

a secure kernel can significantly increase the development cost for a secure Ada application. These workarounds can include modifications to the Ada system software (compiler, RSL, etc.) and the addition of special-purpose hardware and software to implement system security requirements.

10.15 DIVERSITY IN IMPLEMENTATION OF APSE's

The task of developing Ada software for computers integral to weapon systems is a complex one, and would be impossible without good support tools. The set of these tools to be used with an Ada compiler is called an Ada Program Support Environment (APSE), and the APSE has been the subject of much study since the Ada language was specified.

One of the problems recognized very early by both the Army Communications and Electronics Command (CECOM) and by the Air Force was the need for standardized and portable APSEs. The Air Force effort was lost in a funding problem soon after its inception, and the CECOM effort, which resulted in the Ada Language System (ALS), was terminated and made public domain. Sonicraft, which contracted Softech to retarget the ALS to the Intel 8086, tried the ALS and found tool performance below the range of usability.

This left the situation where each compiler vendor has marketed its own version of an APSE, with each requiring training both for users and for the host computer support engineers. This, in turn, has made it far more difficult to transition from one compiler to another during a project, a necessity all too often brought about by other Ada problems [Problem #17 for example].

The extent of this problem depends upon the complexity of the APSE that you now have and the one you are considering acquiring. If one of the APSEs is the ALS, the problem is very severe. Sonicraft had to send three VAX system support engineers to a two week course just to learn to install and configure the tools in the ALS. Users also needed training, which was given by these three engineers. Then, because of the extreme slowness of ALS operations (about a tenth as fast as comparable operations under the DEC VMS operating system), a major effort was required to try to "tweak" the ALS and the underlying VMS parameters to effect a speedup. This big investment in time and money was sacrificed when Sonicraft abandoned the ALS.

Col. Wm. Whitaker (Ret), commenting on a WIS report at the Washington Ada Symposium, stated that most programmers make do with a very minimal support environment, and that many of the exotic tools described in the literature either do not work or are not widely used (or both). One study [Ref] defines the minimal tool set needed as a screen editor and an interactive debugger. You are indeed fortunate if your APSE contains a good source-level interactive debugger [Problem #9], but many APSEs contain tool sets which are quite complex, requiring training for all project designers and host support people.

[Ref] S. J. Hanson and R. R. Rosinski, "Programmer Perceptions of Productivity and Programming Tools", Communications of the ACM, Feb 85

10.15.1 Reliability

The diversity in implementation of APSEs can hurt program reliability during the transition from one APSE to another if it occurs in the middle or near the end of a project. By this time the application has grown complex enough that programmers can become confused, and their unfamiliarity with the new APSE offers just such an occasion. The more complex the new APSE is, and the harder it is to learn, the more severe will this problem become. A confused programmer is more apt to insert a "bug", or program fault, which of course will by definition reduce program reliability until it is found and corrected.

10.15.2 Maintenance

The diversity in implementation of APSEs affects the program maintenance if a switch in APSEs occurs right before delivery. This is not as far-fetched as it may seem, because if system performance meets specifications only marginally, as it did for the Sonicraft MEECN project, an investment in an alternative compiler may be made in parallel with the original one (as Sonicraft has done more than once). If the alternative proves to be better, a switch could be made.

The later in the program this switch is made, the more likely it is that the customer has made some commitments toward system maintenance that will have to be changed. The system maintenance concept is a Critical Design Review (CDR) topic on most programs, so at least some cost is needed to present a new maintenance concept to CDR attendees.

10.15.3 Portability

Diversity in implementation of APSEs can hurt portability in several ways. First, the APSE itself is probably optimized for a particular host, so extensive rework would be required to rehost it.

Second, Sonicraft found it necessary to tailor the application code for both the ALS and the second compiler we used, the Softech Ada Intel Toolset (AIT). Some of these work-arounds, although correct Ada code, may not work on a third compiler, and/or the third compiler may need tailoring of its own.

Third, although the compiler is by far the most complex tool in the current APSEs, some of the other tools may not be portable, even if the host computer is unchanged, because they are strongly tied into the compiler in some way.

10.15.4 Training

Diversity in implementation of APSEs requires retraining of application developers, host computer support personnel, and engineers from other disciplines (Systems, Test, Quality, etc.) who need visibility into the application code each time the project is forced to change APSEs.

10.15.5 Tool Use

Diversity in the implementation of APSEs tends to discourage tool usage, at least for those tools not minimally essential to get visibility into the application code and to be able to manipulate it relatively easily. Sonicraft, having invested heavily in training to use the wide range of tools in the ALS, only to see that investment lost when the ALS had to be dropped, will not soon make another large investment.

10.15.6 Development Environment

Diversity in implementation of APSEs affects the development environment because the user interface to the APSE is usually very different when you go to a new APSE. Although the hardware in the environment does not have to change, it is nonetheless true that the APSE is the controlling mechanism of many of the services that the hardware performs.

10.15.7 Vendor Interface

The diversity in implementation of APSEs affects the vendor interface because the investment in training (especially for a very complex APSE) tends to "lock in" a project to an APSE unless a very serious limitation threatens the project.
In the experience of Sonicraft, this is bad for the project because vendors tend to be much more responsive to project needs if they know they are facing good competition from another APSE vendor.

10.15.8 Compiler

Diversity in the implementation of APSEs limits the ability of a project to switch compilers when an apparently better one becomes available after one has already been used for a while. This is because the investment in APSE training for the existing compiler will probably be useless for the APSE that comes with the new compiler, and additional time and money will be needed for training in the new APSE.

10.15.9 Facilities

Diversity in implementation of APSEs may increase facilities costs if the new APSE requires a different host computer configuration. Sonicraft experienced this when it purchased the Alsys compiler to use to verify correctness of code before submitting a compilation to the ALS. This was necessary because of the ALS very slow compilations and sometimes puzzling compile error messages. The problem was that the Alsys compiler was not available to run on the VAX host that Sonicraft was using, so an additional host computer had to be acquired. (In this case the host was an IBM PC/AT compatible, which was not very expensive and which could be used for other purposes, so the penalty was not severe.)

10.15.10 Stability

Diversity in implementation of APSEs affects project stability from the viewpoint of the customer especially. Although not as drastic as a change in the specifications in the deliverable system, a change in the APSE does affect the maintenance concept and does have some cost due to its change.

The customer is usually willing to change APSEs if it can be shown that a change in specifications to the deliverable system is the only alternative. Other than that, many customers would probably rather have the project stability than a gain in system performance that isn't really necessary.

10.16 POOR PERFORMANCE OF ADA TOOLS

In recent years, a number of software support tools have been developed for use on Ada development efforts. However, due to the fact that most of these tools have been developed for project-specific purposes or as a part of internal R&D efforts, these tools have generally provided poor performance. Some of the tools that have been developed include compilers, linkers, importers, exporters, debuggers, editors, pretty printers, PDL processors, library managers, and library software (mathematical, etc.).

The poor performance of these Ada tools has had an adverse impact in some areas of programmer productivity. The programmers have had to expend significant effort to develop workarounds in those (frequent) cases where the tools have not performed as expected (advertised). The tools vendors have provided poor documentation and inadequate tool support, and a number of the tools have been delivered with bugs in them.

This situation should improve as the Ada software development environment continues to mature and more tools users provide feedback to the tool vendors concerning the performance of the tools.

10.16.1 Efficiency

A number of the currently available Ada tools are immature. The poor performance of these tools can have an adverse impact on overall system efficiency. In particular, the performance of the compiler currently has the greatest impact on system efficiency. The compiler generates the object code for the Ada applications programs and also generates the RSL code, which operates in conjunction with the applications programs.

Two of the major problems to be addressed in the development of Ada systems are the inefficiency of the object code generated by the Ada compiler and the overhead associated with operation of the RSL programs. The object code generated by the compiler runs relatively slowly and uses an extensive amount of memory when compared to the use of other programming languages for embedded real-time applications. The RSL performs a number of run-time functions for the applications programs, but also generates excessive system sizing and timing overhead.

10.16.2 Reliability

A number of the existing Ada software tools are immature. They have not yet been proven by use in actual Ada system development efforts and in some cases still have bugs in them. Since the compiler-generated object code for the applications programs and the RSL both reside in the target machine, any bugs that are present in these programs will reduce overall system reliability.

10.16.3 Benchmark

Because a number of the existing Ada software tools provide poor performance, it is important to perform extensive benchmarking of the tools before deciding whether or not to use them for a particular project. Benchmarking provides an evaluation of the actual performance of a tool in a specific development and applications environment.

Benchmarking also helps to provide an evaluation of claimed (expected) tool performance vs. actual performance.

For example, the execution time required by the RSL to process an interrupt can have a major impact on system performance. A twenty percent increase in the time required to handle an interrupt can adversely impact system performance, particularly if the system processes a large number of interrupts.

10.16.4 Optimization

The poor performance of a number of the existing Ada software tools currently forces the applications developers to perform extensive Ada optimization to meet system requirements. This optimization can include the use of special-purpose hardware and software, modifications to the compiler and RSL, and other techniques. However, while increasing overall system performance, optimization can also have an adverse impact on certain system characteristics. Extensive optimization can reduce system reliability, maintainability, and portability while increasing system complexity.

10.16.5 Development Environment

The poor performance of the Ada software tools that comprise the applications development environment can have an adverse impact on project productivity. For example, the compilation speed of an Ada compiler can make a major difference in the productivity of the development effort. A compilation speed of 100 lines per minute as opposed to 200 lines per minute can cause a significant reduction in programmer productivity, particularly if a large number of recompiles are performed.

10.16.6 Vendor Interface

Due to the poor performance of some of the available Ada tools, and because some of the tools still have bugs in them, it is important to establish an interface with the tool vendors. This interface will allow the vendor to provide support in fixing bugs in the software tools and will also provide a source of information concerning ways to obtain optimum performance from the tools.

10.16.7 Cost

The poor performance of a number of existing Ada tools can have adverse impacts on system performance and programmer productivity, thus increasing the overall system development cost.

10.16.8 Schedule

The poor performance of a number of the existing Ada tools can have an adverse impact on system performance and programmer productivity, thus increasing the length of the development schedule.

10.17 DIFFERENCE IN BENCHMARKING ADA SYSTEMS

Benchmarks can be of two main types, those that are used to time and size portions of application code (usually using breadboard hardware) and those that are used to evaluate compilers and other support tools. It is the benchmark software for support tools that is addressed in this problem.

Ada is a very powerful, but also very new, programming language for embedded, mission-critical software. Whenever an application is being planned that is significantly different from previous experience in the software organization performing the task, benchmarking is normally relied upon to scope out the job. Unfortunately, the very constructs that make Ada a valuable embedded software language are hard to find in widely available benchmarks.

The benchmark most often cited by compiler vendors seems to be the Dhrystone, which has none of the new Ada constructs (tasking, interrupt handling, direct addressing, etc.) which make it superior to languages like Pascal for embedded systems. In comparison to real application code, such as the Sonicraft MEECN system, the Dhrystone benchmark is too optimistic in both compilation speed (lines of code per minute) and in object code size (bytes per line of source code). Errors could be a problem if they are not discovered until application code begins to emerge somewhere around the end of the Detail Design Phase. The project is supposed to be ready to code very heavily at that point, yet will find itself with undersized computer resources, both for the host and the target, if the Dhrystone numbers had been believed.

One approach taken to deal with this problem was reported by M. Kamrad of Honeywell at the Jan 87 SIGAda Conference. He recommended postponing the decision of how many processors to put in the system and what software functions run in each until the coding has matured enough that its final size and timing requirements are known.

According to D. L. Doty [Ref], this can be a long wait. He has observed size-estimate errors greater than 100 percent at the RFP stage, 75 percent up to the Preliminary Design Review, and 50 percent up to the Critical Design Review.

But all this leaves the compiler vendor in a quandry if he is trying to introduce a new compiler. Unless he can test it against a real application in Ada which is already running under another Ada compiler, it will be difficult to convince potential customers that this new compiler can really handle an embedded Ada application.

[Ref] D. L. Doty, P. J. Nelson, and K. R. Stewart, "Software Cost Estimation Study: Guidelines for Improved Software Cost Estimating" (Vol 2), Final Technical Report by Doty Associates, Inc., for Rome Air Development Center (RADC-TR-77-220), Griffiss Air Force Base, NY, Aug 77, 145pp as cited by: W. Myers, "A Statistical Approach to Scheduling Software Development", IEEE Computer, Dec 78

10.17.1 Efficiency

For applications where target efficiency is important, the availability of an appropriate benchmarking program early in the development cycle is very important. Efficiency is needed where the computer resources (memory size and speed of execution) are limited by physical size, weight, power or cooling requirements. In mission-critical software applications it is not unusual to be limited by all these requirements.

The inability to differentiate between compilers that may be suitable for an application, or, worse, false information about compiler performance on that application, can damage the target efficiency actually achieved quite severely. In some cases it may be necessary to buy a second compiler that generates more efficient code.

10.17.2 Maintenance

The use of inappropriate benchmarks to select a compiler, thus defining target code efficiency, and to specify embedded computer resources can lead to a situation where considerable optimization is required to get everything to fit in memory and still execute within timing constraints. Compiler optimization can cause problems if it is unique to a project, causing surprises in the maintenance phase of the life cycle when seemingly straight-forward changes are inserted.

More serious problems occur when obscure source code, especially "tricky" code that depends on a side-effect to work, is used to meet the system requirements. Program maintainability depends mostly on program understandability [Ref], so obscure code is therefore very difficult to maintain.

[Ref] G. M. Berns, "Assessing Software Maintainability", Communications of the ACM, Jan 84

10.17.3 Expandability

The use of an inappropriate benchmark can lead to the selection of a compiler that unnecessarily wastes embedded computer resources and to specifications that are much tighter than anticipated. Even when this does not cause the project to fail by requiring massive rework, the available resources are invariably stretched almost to the breaking point in the process of solving the problems.

In this situation there is simply nothing to spare for future expandability unless the procuring agency had specified unused computer resources to be in the delivered product, and had the further strength to avoid sacrificing this reserve when the project got in trouble because of the inappropriate benchmark programs.

10.17.4 Tool Use

The use of inappropriate benchmarks can ruin the selection of the most important tool on the project, the Ada compiler itself. At the present state of technology there is not often much choice available in compilers, but the differences between them are often quite significant. If the wrong one is selected because it did a good job on an inappropriate benchmark program, the project will usually suffer, especially if it is later proven that the selected compiler has limitations that prevent it from supporting the specified software performance.

10.17.5 Benchmark

The problem subcategory BENCHMARKING here refers to the sizing and timing estimates obtained on real application code of critical routines or algorithms. Usually these benchmarks differ from the deliverable code only in their more thorough treatment of special cases or in exception handling (if they differ at all). The support code benchmarks which are being discussed in this problem treatment are general-purpose routines often obtained from an outside source. The best support code benchmarks are usually widely used in government, industry and academia by designers interested in evaluating Ada compilers.

Inappropriate support code benchmarks can, as discussed elsewhere in this problem, cause application developers to obtain an unsuitable compiler whose performance is over-estimated by the support code benchmark. This must be later corrected by reallocating software functions to hardware (extensive rework), purchase of a higher performance compiler, or optimizing source code. Sometimes it requires some combination of these to recover from the problem.

When the compiler must be replaced this invariably impacts any work that has been performed in application benchmarking since this work is done in the earliest possible design phases. In fact it is often the application benchmarking that demonstrates that the compiler benchmark was inappropriate.

10.17.6 System Sizing

The use of inappropriate benchmarks can lead to errors in target processor memory requirements that can be quite substantial. If the application happens to be one where memory is extremely expensive or even impossible to add (using then-

current technology), this could force an abortion of the project.

Projects with radiation-hardened memory requirements, in which reliability goes down in direct proportion to memory size, and projects with severe size, weight, power and cooling limitations (such as on-board processors in a small missile), are especially vulnerable to memory sizing problems.

10.17.7 System Timing

When a compiler produces more object code than expected (bytes per line of Ada source code), the execution speed of a functional module also increases. Experience at Sonicraft with code generated using a steadily-improving compiler indicates that the relationship is quite direct.

In systems where hardware cannot be added to replace Ada routines that are found to be too slow, and/or where the permissible amount of assembler code has already been used for Ada replacements, this can be disastrous. Use of an inappropriate benchmark, by delaying the time at which the speed problem becomes known, can cause either or both of these situations to occur.

10.17.8 PDL Processor

The PDL processor is often used to help make target memory and speed estimates during the Preliminary Design Phase and especially during the Detail Design Phase. The use of an inappropriate benchmark can throw off the resulting estimates enough to force a substantial rewrite of the PDL to meet system requirements.

10.17.9 Compiler

The use of an inappropriate benchmark can result in the selection of a less-expensive compiler in the mistaken belief that it is good enough to perform the Ada application at hand. In some cases it can even mislead one into thinking that then-current Ada compiler technology is adequate, when in fact an Ada waiver or a substantial reduction of the functionality assigned to software in the Software Requirements Analysis Phase may be needed.

10.17.10 Software Requirements Analysis

The use of an inappropriate benchmark usually leads to overly-optimistic estimates of what can be accomplished in software using the available target computer resources. This can force a project to regress back to the hardware/software allocation of system functions when the truth is finally learned. If this happens fairly late in a project, and/or if the project deadline cannot afford major delay, this would be a fatal blow.

Similarly, cost constraints could preclude the major rework that might be needed.

Benchmarks can also cause the host computer, which is usually selected in this Phase, to be grossly undersized. This could lead to very long turn-around times for compilations unless unplanned host capacity can be added. This kind of activity may not kill a project, but has been known to be detrimental to the careers of those responsible for the error.

10.17.11 Preliminary Design

Errors caused in target memory sizing and target execution times of major functions become part of the Software Requirements Specification if not discovered before the Preliminary Design Phase is completed. Since application code does not appear in significant quantities during this Phase (usually), erroneous benchmark programs can cause considerable damage here.

10.17.12 Detail Design

Erroneous sizing and timing specifications caused by the use of inappropriate benchmarks are often discovered in the Detailed Design Phase, usually when preparing a Critical Design Review demonstration of an early software prototype. As such, the error is usually associated with this Phase, when in fact the benchmarking normally occurs in earlier phases.

A discovery of major specification errors this late in the program costs considerably in both schedule time loss and in rework costs. If the system hardware/software functional allocation must be severely changed, this benchmarking problem could easily be fatal to the project.

10.17.13 Facilities

Inappropriate benchmarks can cause compile times on the host computer selected to be underestimated by very large amounts. Since these most computers often have long lead times, require special power and air conditioning, and are not inexpensive, this can be a serious problem.

If the additional facilities are not purchased, the increased compile turn-around time will reduce programmer efficiency to the point where more staff might be added, reducing efficiency even more.

10.17.14 Cost

The costs that can be incurred by using inappropriate benchmarks to select the Ada compiler and to allocate functions between hardware and software come from two sources:

First, the target computer design may need rework because it has insufficient resources to meet the specifications which the benchmarks had indicated were achievable.

Second, the host computer may be too slow to support the project because the benchmark had given erroneous values of lines of code compiled per minute compared to what was being experienced later in the program when application code was compiled.

10.17.15 Schedule

An inappropriate benchmark can hurt schedule mainly by misleading the designer as to what embedded computer resources are needed to meet the software specifications, perhaps causing rework all the way back to the functional allocations between hardware and software.

A second, usually less important, schedule problem can be caused by believing that the compile speeds attained using the compiler benchmark can be attained using the application code. This causes the host computer to be undersized, giving compile turn-around times that are much longer than may be needed to produce the Ada code in a timely, efficient manner. When compile turn-around gets to be overnight, programmers usually respond by doing more manual checking of submitted code, causing delays in addition to those caused by simple inefficiency.

10.17.16 Estimation

The whole purpose of benchmarking is to estimate the target computer resources and the host computer resources needed to satisfy system design constraints. A benchmark that makes very large errors in doing this is an obvious problem.

10.18 LACK OF ADA SOFTWARE DEVELOPMENT TOOLS

Although a number of software tools have been developed for use in Ada environments, there is still a variety of tools that are desirable to further improve the productivity and performance associated with Ada development efforts. Some of these tools are currently in various stages of development (planning, design. implementation, testing).

Some of the tools that would be useful for Ada efforts include:

* Ada Design Generators
* Ada Code Generators
* Ada Source Code Analyzers
* Ada-oriented Debuggers
* Ada Syntax-directed Editors
* Ada Cost Estimation Models
* Ada Project Management Tools
* Secure Ada Operating Systems
* Automated Ada Test Tools

The lack of Ada tools affects the overall productivity of development efforts. The availability of these tools would decreases the number of development activities that are currently performed manually. It would also reduce the overall development effort by reducing the requirement for applications programmers to develop their own project-specific tools. The availability would also improve the consistency of the products developed on Ada projects and would help impose and enforce project methodologies and development standards.

10.18.1 Reusability

The current lack of Ada software development tools forces the applications programmers to develop their own project-specific tools to perform those functions. However, these tools are generally not built to be reusable in a variety of different applications. This is due to the additional cost of developing a reusable tool as opposed to developing a tool which is only to be used on a particular project.

10.18.2 Development Environment

In an effort to provide a more powerful and flexible environment for the development, a number of Ada software tools have been developed and more are being developed. But due to the immaturity of Ada, there are a number of tools that are not yet available which would prove to be very beneficial for Ada development efforts. Even though these tools are not yet available, some of them are in various stages of development (conceptualization, design, implementation, test). These tools include:

* Ada Cost Estimation Tools
* Ada-oriented Debuggers
* Ada Project Management Tools
* Secure Ada Operating Systems
* Automated Ada Test Tools
* Ada Source Code Generators
* Ada Design Tools

As these tools become available to the Ada applications programmers, a number of the Ada development activities that are currently manual or partially automated will be fully automated.

10.18.3 Cost

The current lack of Ada software development tools forces the applications programmers to develop project-specific tools which perform the required functions. The development of these tools increases the overall cost of the Ada development effort.

10.18.4 Schedule

The current lack of Ada software development tools forces the applications programmers to develop their own project-specific tools which perform the specific functions. The development of these tools can lengthen the overall system development schedule.

10.19 ADA LANGUAGE COMPLEXITY

The complexity of the Ada language makes it difficult to learn, use effectively, and test (validate). The Ada language requires extensive training for programmers to learn language syntax, proposed development methodologies, software engineering standards, and implementation issues for real-time embedded systems.

The Ada language is also difficult to use effectively and efficiently. For example, the current inefficiency of Ada compilers creates an environment where unchecked use of certain Ada features (tasking, generics, etc..) can cause significant impacts on system performance. Also, Ada development methodologies and programming standards/conventions are still being established.

The power and complexity of the Ada programming language can also make it difficult to test and validate an Ada system. The functional and performance impacts of the RSL, a very complex piece of software, play an important part in the testing process.

10.19.1 Verifiability

The complexity of the Ada programming language and run-time environment makes it difficult to verify Ada systems.

While the Ada programming language has a number of features which provide valuable support for state-of-the-art software engineering methodologies, the complexity of the language (packages, tasking, generics, etc..) makes it a very difficult language to verify for program correctness.

In the target environment at run-time, an applications program can perform tasking in a multiprocessor system, create, access, and deallocate portions of memory, and access specific memory locations. Also, the RSL provides a number of run-time functions such as task scheduling, memory management, and others. These functions combine to create a complex run-time environment, in which a large portion of the code resides in the RSL. The complexity of the RSL, combined with the fact that most applications developers are unfamiliar with the operation and performance of the RSL, makes it difficult to verify an Ada system.

10.19.2 Complexity

The Ada language provide a number of language features (packages, tasking, generics, etc..) which support state-of-the-art software engineering methodologies. However, the use of these features greatly increases the complexity of the Ada language and makes the language much more difficult to learn and use effectively.

The complexity of the Ada language, the variety of different compiler implementations, and the extensive run-time overhead associated with the RSL are among the issues that must be addressed by an Ada applications programmer. These issues require that an applications programmer have a good understanding of the Ada language and a working knowledge of the target (run-time) environment in order to effectively use Ada to develop real-time embedded systems. If an applications programmer does not have this knowledge, there is a high probability that the developed system will be inefficient and difficult to verify.

The extensive amount of Ada training that is required for Ada applications programmers is a direct result of the complexity of the language. The training helps to ensure that the applications programmers use the Ada language as it was intended to be used, understand the Ada implementation details, and address critical real-world issues in the system design.

10.19.3 Detail Design

The complexity of the Ada programming language can increase the amount of effort required in the Detailed Design Phase of Ada projects when compared to other languages. The Detailed Design Phase is the point in the project where the proposed Ada implementation approach is developed. It is also the point where the developers begin, in detail, to assess the impact of Ada implementation decisions on system performance.

Due to the complexity of Ada, the resulting detailed design should be reviewed extensively to maximize the effective and efficient use of the Ada language, particularly in light of the system performance problems associated with Ada systems. The applications developers should also identify as many of the system optimization requirements as possible and incorporate them into the system design. These reviews can significantly increase the effort required to develop a detailed design for Ada projects.

10.19.4 Personnel Resources

The complexity of the Ada programming language requires that the applications programmers have previous experience with high-level languages (such as Pascal), and, in particular, languages that are commonly used in the development of real-time embedded systems.

10.20 CUSTOMIZATION OF RUN-TIME SUPPORT LIBRARY

When an Ada compiler and its respective run-time support code is validated, it is on a target system specified by the compiler developer. The problem is that an applications user of the compiler will usually have a different configuration (memory map, I/O map) for their particular system. Also, the user may desire different default values for their run-time application, such as the task stacks. Instead of a four kilobyte default, as experienced on Sonicraft's MEECN project they may elect to have a three kilobyte default. Therefore, the applications user must modify or customize the run-time support code. To get this type of customization the user will need to recompile the run-time support library. If the user desires to perform this task himself, he must usually purchase the source code rights for the RSL and train some people on how to perform the required task. The other option is to contract the vendor to make the required modifications. Either course of action will add more cost to the development of the application program.

Sonicraft discovered on its own Ada projects that the ALS run-time software needed to be reconfigured for memory map allocation, interrupt vector addresses, and Input/Output addresses for Intel's PIC and PIT chips. Further, RSL code had to be repackaged to allow for better selective linking and modified to remove unused 8087 code and to reduce interrupt overhead.

10.20.1 Maintenance

Customization of the Run-time Support Library will increase the cost to maintain the RSL and the application program. For example, additional documentation will be required for any changes required to support the execution of the application program.

10.20.2 Portability

Whenever software is customized to apply to a unique system configuration it sacrifices portability. Thus, the modification of the run-time support software will impact any efforts to import the application to another Ada environment.

10.20.3 Reusability

Due to the customization of the run-time software, the reusability is reduced if the system configuration changes. If configuring the run- time software can be done by setting pragmas in the application source code the loss of reusability could be minimized.

10.21 LACK OF EXPERIENCED ADA PROGRAMMERS

As we in the Ada community have heard so often, Ada is "not just another high order language", but is a whole new approach to software design. Indeed, many respected individuals feel that the major contribution that Ada will make is to train programmers in the use of modern software design techniques [Ref 1]. When we speak of the problem of not enough experienced Ada programmers, it is in this broader context that we see the problem.

Experience at Sonicraft is that a fresh graduate with a CS degree can learn Ada syntax well enough in about three weeks to start making useful contributions to a project. Despite the complaints about Ada being "too complex", it turns out in practice that some of the complex features need only be used by a small percentage of design team members.

Teaching a methodology takes far longer in the experience of Sonicraft. Formal courses typically go for two or three weeks at two or three hours per day (with homework), but it takes actual project experience before most people really understand the value of a methodology and become advocates of it. Without this kind of full support, most methodologies are reduced to being an extra paperwork burden in the minds of the programmers.

Formal courses are another problem because hiring is usually done over a period of time, rather than hiring the first few applicants who meet the minimum qualifications. Putting untrained people on the job while they are waiting for the next class to start brings on the infamous Brooks' Law effects, where the addition of additional programmers slows down the team [Ref 2]. This happens because the experienced people must spend considerable time explaining to the new people some of the things they would normally have learned in the classes.

Finally, the supply of experienced Ada programmers is not easy to measure. At the November 86 SIGAda Conference, for example, it was reported that the biggest shortage in Ada-trained people is among managers. The supply of programmers was greater than the number of people actually doing Ada work. This result flies in the face of experience in hiring at Sonicraft, where over a four year period over forty designers were hired to do Ada work and only one of them had any Ada experience on the job. Very few of the CS majors had a course that even used Ada until about 1986.

This apparent discrepancy between the reported oversupply and the experienced shortage of Ada programmers could have been a local shortage or it could have been pure chance. But it also could have been caused by large firms training massive numbers of people in Ada in anticipation of future Ada contracts. This would indeed cause an oversupply to be measured if one counted

everyone who went through these courses as an experienced Ada programmer.

[Ref 1] F.P.Brooks,"No Silver Bullet", IEEE Computer, April 87

[Ref 2] F.P.Brooks,"The Mythical Man-Month", 1975, Addison-Wesley, Reading, Mass.

10.21.1 Efficiency

When Sonicraft implemented the MEECN Ada project, there was a memory sizing problem discovered after much of the code had been written. By having the most experienced team members go over the code of the less experienced members in code walkthroughs, a very substantial savings in both target code size and run time was achieved.

Ada is a complex language. While it is true that after only a few weeks a programmer will know the syntax well enough to do useful work, the sections of code that require detailed Ada knowledge to make best use of Ada features for efficiency are best left to the more experienced Ada programmers.

This is also a management experience problem, since other high order languages rarely show this much sensitivity to experience. A manager doing his/her first Ada project can easily misassign critical code sections, which then must later be reworked.

10.21.2 Risk

The longer-than-normal delay between hiring an inexperienced Ada programmer and the time that he is "fully productive" (usually defined as spending less than ten percent of the time in training or education-related tasks) increases the risk to projects being done in Ada. Ada projects are hard to estimate accurately [Problem #23]. Unless the developer has trained a pool of Ada programmers, who then may have been assigned to interruptible tasks (such as overhead R&D projects), they may find it impractical to train and then add the required staff in time to meet project delivery dates.

This aspect of the problem is more critical when the project is fairly small, making the duration of each phase only a few months. If an entire phase of the life cycle elapses before an underestimated project is staffed with fully trained Ada designers, the customer will probably consider the resulting schedule slippage a major one.

10.21.3 Training

Inexperience affects training in methodologies because a mind-set is required to use Ada properly. In this regard it is often better to train fresh Computer Science graduates than

experienced FORTRAN programmers who have never learned one of the structured methodologies.

10.21.4 Tool Use

Especially because of the inefficiencies of current Ada compilers [Problem #7], the use of tools like profilers (they tell how often each line or code segment is executed, helping to direct attention to the often innocuous code that is devouring the available computer resources) is important.

10.21.5 Reviews

Because of the complexity of the Ada language, it takes about a year, in Sonicraft's experience, to acquire expertise in the areas needed to write consistently efficient source code. Peer reviews in the form of structured design or code walkthroughs are essential for the work of the inexperienced Ada programmers especially, but if the project has few experienced Ada programmers available they could spend so much time in reviews that little work gets done. This is significant because they regularly produce two to three times as much as new graduates with basic Ada and methodology training.

10.21.6 Method Type

Any new methodology is easier to learn for a person who already is fully trained in the use of at least one other because of the "mindset" problem that needs to be overcome. Experienced Ada programmers are almost certain to be trained in a methodology [Problem #21 Definition], whereas those inexperienced in Ada are frequently not.

Since some methodologies are harder to learn than others (Jackson seems harder than Yourdan, for example), this could be a factor in selecting a methodology if few on staff had ever learned one.

10.21.7 Document

Ada has a number of new constructs that affect what had once been fairly standard concepts in documentation. As an example, how does the concept of Visibility affect the definition of Module Coupling? [Ref] How would you show concurrent tasking on flow diagrams? What symbols do you use to show tasking or packaging?

Even experienced Ada programmers have difficulty answering these kinds of questions, and the problem gets worse with inexperience in Ada.

[Ref] S. E. Watson, "Ada Modules", Ada Letters, vii. 4-79 to 4-84, 1987

10.21.8 Benchmark

The Ada inexperience on a project could invalidate some of the code benchmarking done on breadboard or brassboard hardware because of the unfamiliarity with the more complex Ada features. The area of hardware interfaces is where many of these features are found, so it might be easy to be fooled as to what is really happening during a test.

Sonicraft was forced to assign an experienced Ada software engineer full time to the brassboard to make sure that all benchmarks were done correctly.

10.21.9 Optimization

With embedded computer resources limited by system size, weight, power and cooling constraints, it is to be expected that any delivered Ada code will be compiled with the optimizer turned on. It is true for any language that optimization causes subtle changes to the way things work, but for Ada compilers with little field experience the changes can be more noticeable.

Inexperienced Ada programmers, who may be fully occupied in trying to understand what happens normally, may miss some important changes when optimizing.

10.21.10 System Sizing

Inexperienced Ada programmers do not yet understand the language well enough to pick the most size-efficient constructs available. In the experience of Sonicraft, they are usually satisfied when they find anything that seems to work, and it takes a review by more experienced peers to suggest more efficient approaches.

10.21.11 System Timing

Similar to the system sizing discussion, inexperienced Ada programmers profit a great deal from suggestions from more experienced Ada programmers to speed up their code as required. Often this involves the use of tools, such as the profiler [Problem #21, Tool Usage], with which they may not yet be familiar.

10.21.12 PDL Usage

Even before coding begins the price of Ada inexperience on a project surfaces. Early in the Detail Design phase they are expected to generate documentation using a Program Design Language (PDL), which on Ada projects is normally a compilable Ada syntax with a few key provisions added.

10.21.13 Non-Ada Software

Ada software usually must interface with another language, such as Assembler, in an embedded microprocessor project. Even if they are familiar with the other language, designers inexperienced in Ada are open to additional problems as they try to understand this interface.

10.21.14 **Vendor Interface**

One of the difficulties experienced at first by Sonicraft was the interface to the Ada compiler vendor. As one of the first in industry to attempt a mission-critical weapon system development in Ada, Sonicraft had no one with Ada experience. Since no Ada compilers existed that could meet the system requirements, Sonicraft contracted to Softech to retarget the Army ALS to the Intel 8086 processor.

While Sonicraft engineers understood what they needed to accomplish, having had experience with other embedded microprocessor systems, they found it very challenging to express their requirements in Ada terms.

Obviously the Sonicraft experience was not unique, because one of the issues dealt with in 1983, fast interrupts, was still a hot topic at the January 87 meeting of the Ada Run-Time Environment Working Group (ARTEWG) of the ACM SIGAda.

10.21.15 **Prototype**

The parts of the Ada language learned most quickly by inexperienced Ada programmers are those that are basically Pascal language commands with a slightly different syntax. The kind of commands needed to interface with prototype hardware, however, are not of this variety, and can be a major source of problems to inexperienced Ada programmers.

10.21.16 **Personnel Resources**

One of the hardest things to do for a software project leader is to staff the job with the right number of designers at each Phase of the development cycle. This is true for any real-time project, especially where the software is fairly complex. The lack of a good supply of experienced Ada programmers who can be quickly brought in can make this staffing problem one of the most difficult tasks on the program.

10.21.17 **Cost**

It stands to reason that designers experienced in the language of the application will be, in general, more productive. This is especially true of Ada, where the complexity leads to rich rewards when properly used. The lack of a good base of experienced Ada programmers means not only that these benefits will be largely unattainable for some period of time, but also

that costs must be incurred for training in order to ever realize these potential savings.

Worse yet, mistakes due to inexperience in Ada can be made in the early phases [Problem #21 Software Requirements Analysis] that can increase costs by a factor of two or more if not caught quickly enough.

10.21.18 Schedule

Whenever costs rise unexpectedly, schedule problems naturally follow because staffing requirements go up unexpectedly. If Ada inexperience can cause cost problems, then it also can surely make solution of them very painful: where do you find the experienced Ada programmers you need to increase your staff? And what happens to your schedule if you have to hire inexperienced people and stop to tr... them?

10.21.19 Customer Relations

As noted in the Problem Definition, the lack of experienced Ada programmers is especially severe in the management ranks. Since the procuring agency must assign a manager to interface with the applications software organization, it follows that the agency will have difficulty finding a good manager who also happens to know Ada.

Customer relations are best when there is open communication between the agency and its contractors, and there can be little communication if the agency manager doesn't understand the Ada problems being faced.

10.21.20 Stability

As noted previously [Problem #21, Software Requirements Analysis], the lack of an experienced Ada cadre will probably lead to poor estimates of the embedded computer resources needed to perform software functions in Ada. When this is realized in later design phases, changing the requirements is one of the few options available that are powerful enough to address the serious problems that result.

The instability in requirements, both for hardware and for software, is the direct result of the lack of experienced Ada designers in the early phases, even though this may be hard to recognize. By the time the problem is manifested, the staff has probably gotten fairly good with Ada, so that is not the first problem you might think of when someone asks, "Where did we go so wrong?"

10.22 EXTENSIVE ADA TRAINING REQUIREMENTS

The complexity of the Ada programming language and the system development and run-time environments results in extensive training requirements for Ada applications programmers.

To fully prepare the applications to develop Ada software a variety of training should be provided at different levels throughout the various phases of the project. Training should be provided at a minimum at the following levels - Senior Technical Staff, Junior Technical Staff, and Management.

The training courses to be offered should be selected according to the type and level of personnel being trained. The technical staff members should be offered all or some of the following courses:

* Ada Language Overview
* Advanced Ada Language Issues
* Ada Development Methodologies
* Ada Implementation Issues
* Ada APSE Issues

The management staff should be offered the following courses:

* Ada Language Overview
* Ada Cost/Schedule Estimation and Tracking
* Ada Development Environment
* Ada Productivity Issues

Ada training costs are high and a large amount of time is required to train the programmers. From 3 - 12 weeks should be allotted for the technical staff and from 1-3 weeks for the management staff. The training courses should be scheduled to coincide with the proposed project staffing plan. It should be noted that it is more difficult to retrain non-Ada programmers than to train new programmers.

10.22.1 Training

One of the major issues to be considered in the development of real-time embedded Ada applications is the extensive amount of Ada training that is required for project personnel. This is due to the following reasons:

* Complexity of the Ada programming language and run-time environment
* Current immaturity of the available Ada tools
* The extensive system optimization required to develop a real-time, embedded Ada application
* The current shortage of experienced Ada programmers (those who have actually developed Ada applications)

The required Ada training must be provided to variety of project personnel to include senior technical staff, junior technical staff, and management. Different courses must be provided to the different project personnel to correspond to their experience levels and project responsibilities. The training plans should also reflect the proposed staffing plan and project personnel experience profile.

The training should also reflect the major software engineering issues for the particular application being developed. It is important to include courses which address Ada development methodologies (using Ada as it was meant to be used), implementation issues, and real-world issues (problems/solutions).

10.22.2 Cost

The extensive training required for Ada projects can significantly increase the system development cost. This is due to the requirement to provide a number of different courses (language issues, implementation issues, methodologies, cost/schedule tracking, etc..) for a variety of levels of project personnel (senior technical staff, junior technical staff, management). These courses are generally expensive and, depending on the project staffing plans, must sometimes be given a number of times to coincide with each influx of new people on the project.

10.22.3 Schedule

The extensive training requirements for Ada projects can sometimes have an impact on the project development schedule. This is due to the number of courses that must be offered, the length of the courses, and the number of times that the courses must be offered (to coincide with each influx of new people on the project). To minimize the potential schedule impact of this training, the required training time should be included in the project schedule estimates.

10.23 INACCURACY OF C/S ESTIMATE FOR ADA PROGRAM

Most of the Ada problems recounted here cause cost/schedule perturbations for embedded mission-critical applications [Problems #1,3,4,5,6,7,8,9,10,11,13,14,15,16,17,18,19,20,21,22,23,24,25,26].

Because of the pervasiveness of the influence of almost every problem on cost and schedule performance, the task of estimating cost and schedule performance becomes even more complicated. Not only must you understand enough about software cost/schedule estimating, but you must also understand enough about the additional problems you get in Ada. This Ada problem knowledge is a very rare commodity today, which is one of the reasons this report is being written.

In an effort to help, there have been extensions to one of the most popular cost models, COCOMO [Ref 1], that attempt to account for some of the better-known Ada problems [Ref 2]. These include the instability of the compiler (major revisions every six months being the present norm) and the extra time it takes to train Ada programmers.

Even someone who has lived through the Ada problems might be hard-pressed to estimate their cost/schedule effects on a new project. The only real way to do this is to have someone in control of the project who understands the pitfalls well enough to avoid them, making their cost/schedule effects near zero (since avoidance costs are usually small).

But not all Ada-specific cost/schedule factors can be called problems. After all, the ultimate reason to use Ada is to save money, not to lose it. When a steady-state condition is reached after a few years of using Ada, developers should find the costs much improved. But again, since very few have reached this state, the actual gains are very difficult to estimate.

But even when all Ada-specific cost/schedule influences can be accounted for, which may be years in the future for many applications developers, the job of software cost/schedule estimating is anything but easy. This is a long-standing software engineering problem, with even estimation models with many years of good service being criticized for making errors of 100 percent or more [Ref 3]. In fact, some recommend collecting your own statistics for several project done using your own programming support environment rather than using the factors in the models [Ref 4]. Published model factors are based on data from hundreds of projects, but if your environment is significantly different than the industry norm then your responsiveness to these factors may indeed be unique.

[Ref 1] B. Boehm, "Software Engineering Economics", Prentice-Hall, Englewood Cliffs, NJ, 1981

[Ref 2] R. W. Jensen, "Projected Productivity Impact of Near-Term Ada Use in Software System Development, "Hughes Aircraft Co., Fullerton, CA 1985

[Ref 3] C. Kemerer, "An Empirical Validation of Software Cost Estimating Models", Communications of the ACM, May 1987

[Ref 4] R. Davis, "Measuring the Programmer's Productivity", Engineering Manager, February 85

10.23.1 Risk

The inability to accurately estimate the cost/schedule requirements of an embedded Ada application significantly increases the technical risk that the job may never be finished at all. This occurs when the job is grossly underbid, even in a cost-plus environment (in fixed-price, the job will probably be aborted as soon as the problem is realized).

The first response of most developers to an underbid job is to respond in ways that at least offer a possibility of maintaining schedule commitments. This may include subcontracting modules, hiring job-shoppers to work in-house, and trying to add more hours from the dedicated project staff (through hiring and overtime).

Doing any or all of these things makes project communication very difficult. In the absence of a strongly disciplined development methodology which all but forces this communication, it probably just won't happen. Unfortunately, anything that has the appearance of a non-crisis task tends to get buried in the panic environment of a badly underbid project, and methodology requirements are definitely non-crisis.

Without good *communication* the chances of a complex project ever completing are not good. If it does complete, it will probably need an "update" immediately, where update is a euphemism for "start over and do it right this time."

10.23.2 Personnel Resources

The inability to estimate cost/schedule requirements accurately will impact personnel resources first. Sonicraft, working on the MEECN project, found just after the Preliminary Design Review that the staff needed to quadruple to meet the Critical Design Review date. A combination of temporary help (job-shoppers), hiring and heavy overtime for six months brought in the detail design acceptably close to the original date. While this prevented a quick cancellation of the job, it by no means

solved the staffing problem, which continued in one form or another for two more years, mostly due to the inability to forecast accurately.

10.23.3 Facilities

Closely related to cost/schedule forecasts are the facilities needed to support the development staff. Underestimating the staff needs can be partially compensated by adding facilities that boost productivity, and many companies use this tactic (along with others in parallel). The reason that this is a problem is that the productivity gains rarely justify the cost of such emergency purchases, and facility work on other projects with higher savings must be postponed.

If forecasts are really far off (and that is not at all uncommon), the added staff may find present facilities inadequate. Contention for available host computer time, for breadboard test time, and even for open host computer terminals may slow down the project to the point where the added staff has little effect.

10.23.4 Customer Relations

The inaccuracy of Ada cost/schedule estimations can be counted on to erode whatever good relations were previously enjoyed with the customer. No customer likes dealing with a contractor who regularly produces unpleasant surprises affecting the cost, the delivery date, and (when desperate) the specifications of the delivered system.

10.23.5 Stability

The inability to estimate Ada cost and schedule leads to many short-term responses which attempt to minimize schedule slippage. As schedules are seen to start slipping anyway, and as costs start to climb alarmingly, any good manager will look to the only relief in sight: the specifications.

In any system there are "nice-to-have" functions which become the focus of intense negotiations when a project is in trouble. Since the contracting agency doesn't want to see the project fail, this usually results in some contract reinterpretations or modifications that give some relief to the contractor.

When it happens, the unfortunate result of this is an instability of the specifications that affects not only the two parties directly involved but also many support organizations and, eventually, the using and maintaining organizations.

10.24 LACK OF ESTABLISHED ADA SOFTWARE DEVELOPMENT METHODOLOGY

One of the most important features of Ada is that it encourages the use of modern software engineering development practices [Problem #21]. These practices can best be exploited when they are packaged within a good development methodology. The lack of an established Ada development methodology can be a problem in that engineers trained in one methodology are less mobile if other organizations (even within the same parent organization) use a different one. This in turn contributes to the lack of experienced Ada programmers [Problem #21] since an important part of their experience is their methodology training.

This is not to say that there is a lack of good Ada methodologies. In fact there are two very good ones, Object Oriented Design (OOD) and PAMELA (TM George Cherry). One of these, OOD, is gaining both popularity and respect very rapidly, and PAMELA, while held back now by limitations described below, has good long-term potential also.

OOD is predicated upon the premise that problem definition is the first and hardest task a designer confronts. OOD concentrates on helping the designer with this task by stripping away much of the syntactic material that adds little to this task. It does this by defining the Object as an Ada package or task, which is a higher level view than the module of Yourdan [Ref 1]. This has led some to proclaim OOD as being the most promising of the "technical fads" now available to improve programmer productivity [Ref 2]. But it also causes some to complain that OOD is too limited when it comes to expressing the dynamism of Ada systems in operation [Ref 1].

PAMELA, on the other hand, is best at describing the dynamism of Ada systems. It is the first process-flow methodology specifically adapted to the Ada syntax [Ref 1], using easy to learn graphical techniques to input the design information that it will use to generate code automatically. This labor-saving step is also its biggest problem right now, however, since it generates an Ada task for every "single-thread" (no children) process [Ref 1]. With current Ada compilers the overhead due to context switching is too high to allow free use of tasking.

Sonicraft, in the MEECN system development in Ada, was forced to redesign extensively to cut down the number of tasks in the system because it could not meet the design constraints. Even the delivered design, with only ten tasks, was a problem because some were nested as much as three levels deep, causing very high overhead with the compiler then in use. This is the other objection to PAMELA: it leads to deeply-nested structures [Ref 1].

Note that neither of the two limitations to the use of PAMELA are necessarily long-lasting ones. When Ada compilers become available that can do very rapid context switches (perhaps in

conjunction with underlying microprocessors that have richer instruction sets) and that maintain a single task stack for the parent task and all direct descendants (to avoid context switches among them), PAMELA may become much more widely used.

As a final note, there are some software organizations that service customers outside the Department of Defense and thus may need to use languages other than Ada on some projects. Sonicraft, for example, is currently involved in the FAA's Airport Modernization Program, and was specifically directed to use the same language as the Principal Contractor, which happened to be "C." Since one of the main goals of a functional software organization is the balancing of labor resources between on-going and planned projects, the use of a non-language-specific methodology was preferred. This allowed programmers to be moved between the two major projects to satisfy special project needs or to further individual career goals. Had Sonicraft faced major retraining costs, because the methodology was changed as well as the language, such moves would have been far less frequent.

[Ref 1] S. Boyd, "Ada Methods: Object-Oriented Design & PAMELA", SIGAda, Nov 86

[Ref 2] F. P. Brooks, "No Silver Bullet", IEEE Computer, Apr 87

10.24.1 Method Type

The lack of an established Ada software development methodology makes it harder to decide which one to use on a specific project. There had been a similar problem with software written in other languages until the Yourdan methodology of Structured Analysis and Structured Design "became pervasive within the software engineering community" [Ref]. The dangers inherent in picking a methodology that turns out to be off the mainstream of the community include:

* Extensive retraining will be needed for new hires, who will be unlikely to know your methodology

* Communication with other projects in Ada will be crippled because you will probably use terms like "module" to mean something different than they might define them to mean

* Performance may be hard to relate to industry norms because you might be measuring different things within your methodology, or you may have an inappropriate set of tools to measure what others do

[Ref] S. E. Watson, "Ada Modules", Ada Letters, vii. 4-79, 1987

10.24.2 Personnel Resoures

The lack of an established Ada software development methodology makes the Ada designers less mobile between projects or organizations. When extensive methodology training is required to introduce new members to a team, the project leaders tend to exhaust other possibilities first, such as massive overtime.

10.24.3 Facilities

Methodologies tend to have standard tools and facilities that make their use more efficient. If a standard Ada methodology were to emerge, so that managers knew that any investment in these tools would also pay off for future projects, they would be more willing to buy them.

This willingness to purchase the tools would help create a mass market for them, driving down the price and improving both the quality and the features as more vendors were attracted to serve the burgeoning market.

10.24.4 Cost

The costs associated with the lack of an established Ada methodology are those which are incurred when someone already trained in one methodology must be retrained in another. These costs would be reduced if most projects used the same or very similar methodologies.

10.24.5 Schedule

The schedule delays which occur due to the lack of an established Ada methodology are those incurred due to the length of the training time when adding new staff. Instead of being productive in a few weeks, when the basics of Ada syntax and the basic tools on the host computer have been learned, the new designer requires additional weeks or months to get comfortable with the methodology being used.

An established methodology would reduce these delays by increasing the probability that the new designer already knows the methodology being used.

10.24.6 Estimation

The usual problems in estimating a software job are compounded if there is no established Ada methodology. Since planning mistakes are usually rectified by adding new staff (assuming the project duration is long enough to tolerate this), anything which increases the cost of adding staff will have a leveraging effect on cost estimates, making even relatively small errors more expensive to correct (both in cost and in schedule delays).

10.24.7 Customer Relations

Customer relations can suffer from the lack of an established Ada design methodology in two ways. First, and most important, the customer may be unfamiliar with the methodology being employed. This hurts the needed communications since output and progress may be measured in ways the customer does not fully understand. If the customer has monitored other projects that have used the contractor methodology, the relationship becomes more comfortable.

The second problem is that it becomes a little more expensive to do things if there is no established methodology. Project cost and schedule can increase [Problem #24 Cost, Schedule], and it also takes more effort for the customer to learn effective ways to monitor the work.

10.24.8 **Stability**

In extreme cases a project can be subjected to massive cost and schedule overruns. If the product is deemed valuable enough, the project may be allowed to continue, which then makes the lack of a standard Ada design methodology important. By raising the cost of training the additional staff needed to recover, plus extending the time needed for them to become trained enough to contribute, this problem could force both the customer and the contractor to look for ways to down-scope the product specifications.

The resulting instability in what the product is expected to be will cause a whole new set of problems.

10.25 LACK OF ESTABLISHED ADA SOFTWARE STANDARDS AND GUIDELINES

Sonicraft is among the growing number of software development organizations that measures the productivity of each designer and manager, rewarding them primarily on this measured productivity and the quality of their software products. This has been proven over and over to be a sound practice since we have observed that the difference in productivity between the best and the worst programmers to be consistently about a factor of ten. Besides keeping turnover of the best people very low, this policy also strongly encourages the worst to either quickly improve or to find another line of work.

The reason all organizations don't use this practice is that it takes a considerable amount of effort to make it work right. The measurands we use are, by now, quite standard (operational lines of code and hours charged to the project), with good tools available to automate their collection. The hardest part is to establish standards that can be anchored in some facts, such as the ubiquitous "national average productivity", or a baseline formed from several similar projects done in our environment [Ref].

Here is the problem that Ada introduces. Because of the difficulties in estimating the number of lines of code or labor hours that a project should take [Problem #23], the credibility of the standard is jeopardized. As long as the standards were based on measured accomplishments of others, they were accepted as a challenge. But if they must be based on theory because Ada has a dearth of real data, their motivational value is greatly reduced.

Compounding the problem is the observation [Problem #23] that the productivity of Ada programmers is likely to start out worse than with other languages, then rapidly improve, ending up with higher productivity than other common languages. This makes productivity data collection far more difficult, since it is continually changing over a period of two or three years. It also makes the data from other organizations more suspect since it is hard to tell exactly where on the learning curve they might be operating.

[Ref] H. Davis, "Measuring the Programmer's Productivity", Engineering Manager, February 85

10.25.1 Personnel Resources

The lack of established Ada productivity standards can cause higher turnover among the most productive people. These people can no longer be rewarded as well when the standards are uncertain or poorly defined. Similarly, the least productive people will be less inclined to improve, preferring to blame their troubles on the questionable measurement standards.

The net effect of these two tendencies is an overall lowering of the project team productivity. This strains the available personnel resources in support of the project, which, because of other problems [such as Problem #21] may be difficult to replenish.

This effect can be startling in its magnitude when seen in action on two apparently similar projects, the main difference being the use or non-use of productivity standards. It is possible to see the project using the standards operate at double the national average productivity while the non-user operates at half.

10.25.2 Cost

Due to the combined effects of losing more of the most productive designers and being saddled with non-improving poor performers [Problem #25 Definition], the unavailability of established Ada productivity standards can drive up project costs very substantially.

Disturbingly, the process doesn't take long once it is set in motion. A project can get a reputation as being "a bunch of losers" after only a year or two, which will hasten the departure of the best people still remaining.

10.25.3 Schedule

The combination of rising amounts of labor needed to do each task [Problem #25 Cost] plus the difficulty of replenishing the available labor base in support of the project [Problem #21 Personnel Resources] can lead to missed deadlines.

10.25.4 Estimation

The lack of established Ada productivity standards can increase the turnover of the most productive designers on the team [Problem #25 Cost]. Since these are the people who get most of the work done, the increased risk of losing them makes the estimation process even more difficult.

The reverse problem also occurs. The lack of established Ada productivity standards can decrease the rate of improvement or elimination of the least productive designers. Carrying a poor performer not only increases cost but makes it more difficult to add a more productive person to the team. This also makes the process of estimation more difficult by weakening one of the main sources of management control: when things start going poorly they can deteriorate very quickly, and a rapidly-changing process is almost impossible to estimate.

10.26 PRODUCTIVITY IMPACTS OF ADA

Of the preceding 25 problems, 18 have been shown to influence the cost of an embedded Ada system. On the other hand it is often true that problem identification is the hardest part: once you know what the problem is, the solution is sometimes obvious. This report is therefore not as discouraging as it may seem at first reading.

In the case of Ada, there are some long-term cost benefits that are the ultimate reason it has been adopted as the standard language for Department of Defense embedded mission-critical software [Ref 1]. And other languages are not problem-free: MGEN Smith reported that 70 to 80 percent of the late Air Force projects are having software problems [Ref. 2]. At the same conference MGEN Salisbury reported another need for a standard language like Ada that can handle a wide variety of applications: The Standard Army Management Information System (STAMIS) had grown to 750 systems, now reduced to 110. Using Ada is expected to cut it to 37 programs.

The one area which is probably the biggest source of Ada productivity problems is the speed of the compiler. Compilation speeds have been steadily dropping for VAX-host, Intel-target compilers. Sonicraft has seen a times ten improvement here in the last two years, with another big improvement reported to be in the offing for a compiler to be validated in September 1987 [Ref 3].

For the present, however, Ada does have serious productivity impacts which can price it out of the market for many projects IF YOU LOOK AT DEVELOPMENT COSTS ONLY.

[Ref 1] Department of Defense Directive, 3405.2, Subject: Use of Ada in Weapon Systems, Mar 87

[Ref 2] Policy Committee Reports From Armed Services, SIG Ada, Nov 86

[Ref 3] L. Silverthorn, DDC-I, Phoenix, AZ

10.26.1 Debugger

The debugger currently available for the Intel microprocessors was derived from the Intel Pascal debugger and cannot follow the context switches of Ada tasks (since nothing comparable exists in Pascal).

Debugging within the scope of a single task does not sound overly restrictive, but many of the most difficult code problems involve interrupts whose handlers are implemented as tasks. This limitation can seriously impact productivity during the Code and Test Phase and the CSC Test Phase.

10.26.2 PDL Processor

There have been a number of improvements since Sonicraft wrote its original MEECN Project CPCI Product Specification (C5) in 1983 and early 1984. The PDL processor used was incapable of compiling the Program Design Language (PDL), which caused severe rework problems that could have been largely avoided.

The current PDL processors are still being actively developed, and the discussion continues as to exactly what functions a PDL processor should support. When standardization occurs the productivity on Ada work will rise somewhat, although this is no longer the problem it once was.

The directive mandating the use of Ada [Ref] does encourage the use of compilable PDL, which is the right way to go in Sonicraft's experience.

[Ref] DoD Directive 3405.2, "Use of Ada in Weapon Systems," March 87

10.26.3 Compiler

The Ada compiler, despite tremendous improvements in the past few years [Problem #26 Definition], is still the source of much of the lost productivity faced by an embedded Ada software developer. When you can read about a "C" compiler that is advertised [Ref] to run about a hundred times as fast on a plain-vanilla Personal Computer clone as the best Ada compiler can run on a VAX, you know there is room for improvement here. To be fair, the Ada compiler does much more than any "C" compiler, but the difference should not be a factor of a ten to hundred when far less computer resources are available to it.

[Ref] Turbo "C" advertisements for 10,000 lines of code per minute compile speed

10.26.4 Personnel Resources

With personnel resources already stretched thin by the lack of experienced Ada programmers [Problem #21], the poor productivity of Ada tools can really hurt a project, especially if it comes as a surprise.

10.26.5 Facilities

At one point in the MEECN Project development, Sonicraft seriously considered the purchase of a "RAM Disk" to improve the then unacceptably slow speeds which were available for VAX-host, Intel-target cross- compilations. Fortunately an appreciably-faster compiler was released before this commitment was made.

For some projects, in other circumstances, decisions to upgrade facilities to compensate for poor Ada tool productivities may be required.

10.26.6 **Cost**

Cost is directly dependent on productivity, so low productivity Ada tools will increase Ada software development costs.

10.26.7 **Schedule**

Schedule problems could arise from the poor productivity of Ada tools only if they are unanticipated. This is very possible, since the benchmarking software for these tools is still inadequate [Problem #17].

10.26.8 **Estimation**

The low productivity of Ada tools makes the estimating task more difficult simply because a larger effort is required. When the project staff goes up, the interactions internal to the staff as well as those with elements outside the project staff get more complex.

A more serious effect on estimating, however, is the uncertainty as to exactly how productive the Ada tools are [Problem #17].

10.27 IMPACT OF CONSTRAINT CHECKING ON SYSTEM PERFORMANCE

Ada provides the capability, within the language, to perform constraint checking. Constraint checking provides the Ada application developer with a means to determine whether a variable was assigned a value at runtime that is outside of its defined range. When this event occurs, the exception CONSTRAINT_ERROR is raised to signal the problem. "However, the event that the requirements for constraint checking become too severe, Ada provides a SUPPRESS pragma to disable this feature."

10.27.1 Efficiency

The inclusion of constraint checking can adversely affect the efficiency of an Ada application. The overhead associated with performing constraint checking can cause thee application to use more CPU resources.

10.27.2 System Sizing

The constraint checking code resides in memory as part of the runtime environment. Even if the SUPPRESS pragma is used to disable constraint checking, the unnecessary code still resides in the runtime environment. If constraint checking is not desired for an application. the unnecessary cou be removed from the runtime.

10.27.3 System Timing

An application which performs a large amount of constraint checking could experience a significant increase in system timing overhead. This overhead is in addition to the normal range checking that is performed by a developer within the application program itself.

10.28 INABILITY TO ASSIGN DYNAMIC TASK PRIORITIES

"Ada does support a capability for dynamically altering the priority of a currently running task. The value for the pragma PRIORITY is static and therefore cannot be changed at runtime. Implementations may support an alternate set of priorities that control tasking in the case where the Ada PRIORITY is identical or undefined. ... This allows an implementation-defined subpriority, which may be dynamic, to control the scheduling. This capability is not supported by many implementations, and a standard does not exist to help provide commonality."

10.28.1 Complexity

Because of the inability to assign dynamic task priorities within the Ada language, the developer must implement this feature if it is desired. The developer can implement a limited form of dynamic priorities with considerable effort and at the cost of a considerable increase in the complexity of the application.

10.28.2 Portability

To implement the assignment of dynamic task priorities, the applications developer must have knowledge of the dynamic defined subpriority as implemented in the version of the Ada compiler that is being used. Using this knowledge, however, reduces the portability of the Ada application by building in implementation dependencies.

10.29 INABILITY TO PERFORM PARALLEL PROCESSING

10.29.1 Efficiency

For those operations which lend themselves to parallel processing, the efficiency with these operations are performed could be greatly improved through the use of parallel processing. An example of this type of application would be matrix arithmetic operations.

10.29.2 Complexity

The implementation of parallel processing-oriented operations in a non-parallel processing-oriented environment can cause a significant increase in system complexity. This could involve the use of special-purpose hardware and software to perform the parallel processing operations.

10.29.3 Portability

The use of special-purpose hardware and software to implement parallel processing operations can reduce the portability of the application.

10.30 LACK OF SUPPORT FOR LOW LEVEL OPERATIONS

"Ada does not provide a mechanism to control the processor state (including interrupt masks required for critical sections). Although Ada provides a mechanism to directly manipulate memory mapped hardware, no capability exists within the language to access internal processor registers. Such a mechanism would be difficult to standardize."

For example, "..changing the processor state needs to be done in conjunction with the runtime. Since stacks used for different states are often separate, simply changing state will result in an error condition. Also, subsequent calls to the runtime (possibly due to exceptions) are likely to cause unpredictable results."

Another example is that "..there is frequently a need to enable and disable interrupts which is performed by setting or clearing interrupt masks. It is easy for a programmer to write an assembly language routine to manipulate an interrupt mask and call this routine from an Ada program. The problem occurs because the assembly language is not working in conjunction with the runtime environment provided."

10.30.1 Complexity

To perform low level operations within an Ada application, the applications developer must try to account for for any possible impacts to the RSL. The implementation of these operations can significantly increase the complexity of the application.

10.30.2 Reliability

The overall reliability of an Ada application can be affected by the potentially unpredictable results that can occur when low level operations are not implemented in conjunction with the RSL.

10.30.3 Correctness

The unpredictable results that can occur when low level operations are implemented can adversely impact system correctness by making it difficult to ensure that the system performs as specified and required.

10.31 INABILITY TO PERFORM TASK RESTART

Applications which require that a separate thread of control (task) be restarted at the beginning after being interrupted part way through have difficulty mapping this requirement to Ada.

"Certain applications do have a need to be able to have multiple tasks, where one task might be pre-empted by a higher priority task, and the result of the pre-emption is to make the continuation of the pre-empted task meaningless."

"The standard Ada solution to this problem is to ABORT the pre-empted task, and then re-activate a new task. This creates a few undesirable side effects, not the least of which is likely to be unacceptable performance degradation."

10.31.1 Efficiency

Performing a task restart by aborting the pre-empted task and re-activating a new task can adversely affect efficiency by causing performance degradation. The performance degradation results from the overhead associated with task activation and deactivation.

10.31.2 Reliability

The overall reliability of an Ada application can be affected by the potentially unpredictable results that can occur when aborting a task and restarting a new task.

In the book "Software Engineering With Ada", Grady Booch says: "This [aborting a task] has the effect of prematurely killing a task and all of its dependent tasks. This is a rather ungraceful means of task termination and should be done only when all other means fail."

The task abort and restart are also somewhat non-deterministic; a task is ACTUALLY started or stopped at some time after the request is made. This time interval depends on the implementation of tasking in the particular Ada environment being used.

10.31.3 Correctness

The unpredictable results that can occur when tasks are aborted and restarted can adversely impact system correctness by making it difficult to ensure that the system performs as specified and required.

10.32 INABILITY TO PERFORM CYCLIC SCHEDULING IN ADA

Cyclic scheduling provides the capability to perform periodic processing by running a number of processes on a scheduled time basis. "The Ada language can support some degree of periodic processing by using the DELAY statement. Although some implementations provide a reasonable mechanism for this, the DELAY statement is not always adequate for this application."

"The problem [with the DELAY statement] is that the duration value is a delay from the current time, not a fixed interval. Therefore, the clock must be read and the cycle computed in the simple_expression allowed for the DELAY statement. However, there is no way to ensure that an interrupt (and possibly a higher priority task) is not executed between the time the clock is read in the simple_expression and when the delay duration is actually interpreted by the runtime [program]."

10.32.1 Correctness

The non-determinism of the Ada DELAY statement in the implementation of a cyclic scheduler has an adverse impact on correctness. This is because the applications developer cannot ensure that the period that is specified for running a process (timeslicing) is actually the amount of time that the process runs.

10.32.2 Complexity

The difficulty in trying to implement an accurate cyclic in Ada using the DELAY statement can cause a significant increase in the complexity of the Ada application which contains the scheduler.

10.33 LACK OF FLOATING POINT COPROCESSOR SUPPORT

"A floating point coprocessor is a high performance numerics processing element that extends the main processor architecture by adding significant numeric capabilities and direct support for floating point, extended integer, and BCD data types. The presence of a floating point chip would increase performance in a real-time embedded application that required floating point operations to be performed."

"There is a lack of a standard for floating point coprocessor support in Ada. Some compilers require a floating point chip to perform floating point processing; other compilers cannot utilize the chip if it is present."

10.33.1 Efficiency

The lack of a floating point coprocessor means that more of the system resources (CPU time, memory) will be required to perform floating point operations and to provide support for intrinsic functions such as sine and cosine.

10.33.2 Correctness

The lack of a standard for use of a floating point coprocessor means that some Ada compilers will require the presence of a coprocessor and some will not. Depending on the manner in which the floating point coprocessor is used and the way in which floating point is implemented by a particular compiler, an application could provide different answers for the result of a floating point operation.

10.34 INABILITY TO RECOVER FROM CPU FAULTS IN ADA

"CPU fault tolerance is the built-in capability of a system to provide continued correct execution in the presence of a limited number of hardware or software faults. Highly reliable systems require that the software continue to operate in the presence of CPU faults."

"Although this may seem impossible, careful analysis indicates that many faults are momentary and do not result in permanent interruption of processing capability. However, it is essential that the program be able to recover from such faults and continue execution from the last check point. Ada does not directly support the ability to recover from such CPU faults."

10.34.1 Reliability

"Although it is possible for an Ada program to checkpoint its data, due to the complexity of program elaboration, it is difficult for an off- the-shelf runtime to roll back and recover from a CPU reset." Thus, the unpredictability of the CPU fault recovery process can significantly reduce the overall reliability of the system.

10.34.2 Correctness

The ability to dynamically create objects in Ada at runtime (data, tasks, etc...) and the variety of dynamic objects that are created by Ada during program operation make it very difficult to perform the reconstruction of the runtime environment that is required to properly (correctly) recover from a CPU fault and continue execution from the last check point.

10.35 IMPACT OF ADA COMPILER VALIDATION ISSUES

"Validation is the process of checking the conformity of an Ada compiler to the Ada programming language [as specified in MIL-STD- 1815A] and of issuing certificates indicating compliance of those compilers that have been successfully tested. It should be emphasized that the intent is only to measure conformance with the standard. Any validated compiler may still have bugs and poor performance, since performance is not being measured by the validation tests."

"To obtain a validation certificate, a compiler implementor must exercise an Ada Compiler Validation Capability (ACVC) test suite. The current level is Version 1.9 and it contains a series of over 2500 tests designed to check a compiler's conformance to the DoD's Ada language standard, ANSI MIL-STD-1815A-1983."

"With the initial validation phase completed for most compilers, the compiler implementors are [finally] shifting their emphasis to concentrate on improving the efficiency of the generated code (code optimization) and providing more user configurability of the runtime environment."

10.35.1 Efficiency

The efficiency of the code generated by the current Ada compilers has not been very good because the emphasis has been on passing the ACVC tests to achieve validation and not on achieving the performance levels to support the development of real-time embedded systems.

10.35.2 Portability

As the compiler vendors shift their emphasis to improvement of compiler performance, it expected that one approach will be to address more of the machine dependencies and implementation details. However, taking advantage of these machine dependencies and implementation details will reduce the portability of the code produced by the Ada compilers.

10.35.3 Stability

The compiler validation process can affect project stability. One issue has "appeared involving what constitutes ""maintenance"" of a compiler and how much of it [the compiler] can undergo change and still retain validation status.

"Also, since Ada validation status is only retained for one year after validation, concerns have been expressed for programs that do not want to change the version of their compiler after they begin testing. New policies have been developed to support baselining a compiler with respect to a project, and deriving validation status for similarly

configured machines."

10.36 INABILITY TO PERFORM ASYNCHRONOUS TASK

"The Ada rendezvous model uses a synchronous mechanism to communicate between tasks. Many applications require that a signaling task not be delayed until the signaled task is ready to accept the signal. The mechanism used to communicate between tasks in the Ada rendezvous model is that both tasks must be synchronized together before any data or control information can be transferred."

"The Ada solution to this issue is to place an intermediate task between the signaling task and the waiting task. This intermediate task would always be ready for a rendezvous and would effectively buffer the transaction to provide asynchronous communications. The impact is to create an additional (logical) context switch."

10.36.1 Efficiency

The overall system efficiency is reduced due to time wasted because a signaling task is delayed until the signaled task is ready to accept the signal.

If asynchronous task communications are implemented through the use of an intermediate task, the additional context switch that is required (due to the inclusion of the intermediate task) can significantly increase the overhead associated with this activity.

10.36.2 Complexity

Any optimization that is required to compensate for the possibly extensive waiting time for a signaled task to accept the signal could increase the overall complexity of the system.

10.37 LACK OF IMPLEMENTATION OF THE IMPLEMENTATION

"Many of the features in Chapter 13 [of the Ada Reference Manual] are not implemented in current commercially available compilers today. Chapter 13 of the Reference Manual for the Ada Programming Language is titled, ""Representation Clauses and Implementation- Dependent Features"". These features are optional and therefore a compiler can have the status of ""validated"" without any of these features implemented. However, many people feel that Chapter 13 is required for real-time embedded applications."

The features addressed in Chapter 13 of the Ada Reference Manual allow an Ada application developer to perform systems programming tasks by providing a physical representation of the underlying machine. These features include:

* Representation Clauses
* Length Clauses
* Enumeration Representation Clauses
* Record Representation Clauses
* Address Clauses
* Address Clauses For Interrupts
* Change Of Representation
* The Package SYSTEM
* System-Dependent Named Numbers
* Representation Attributes
* Representation Attributes Of Real Types
* Machine Code Insertions
* Interface To Other Languages

10.37.1 Complexity

The use of an Ada compiler without the Chapter 13 features implemented increases the complexity of the Ada application because the developer must build these interfaces to the underlying machine. These interfaces must also be built to operate in conjunction with the runtime environment.

10.37.2 Non-Ada Software

The requirement for the Ada applications developers to build interfaces to the underlying machine causes them to use a higher amount of non-Ada software.

Maintainability -The additional use of non-Ada software and the required interfaces to the runtime environment tend to decrease the overall maintainability of the system.